OULUN AMMATTIKORKEAKOULU

Lasse Kangas

# DATAN VISUALISOINTI REAALIAJASSA PYTHONILLA

# DATAN VISUALISOINTI REAALIAJASSA PYTHONILLA

Lasse Kangas
Opinnäytetyö
Syksy 2023
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

**TIIVISTELMÄ**

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Lasse Kangas
Opinnäytetyön nimi: Datan visualisointi reaaliajassa Pythonilla
Työn ohjaaja(t): Manne Hannula
Työn valmistumislukukausi ja -vuosi: Syksy 2023 Sivumäärä: 28

---

Opinnäytetyön aiheena oli kehittää Nokian testiautomaatioon reaaliaikainen mittausdatan hahmottaminen. Toimeksiantajana toimi oululainen Nokia, joka keskittyy telekommunikaation tuottamiseen ja kehittämiseen. Työn tavoite oli saada testiautomaation käyttäjille nopea ja selkeä mittaustulosten esittäminen kuvaajassa.

Toteutus tehtiin Python-ohjelmointikielellä Nokian testiautomaatioon. Opinnäytetyöhön kuului komponentin visuaalinen ja ohjelmistoarkkitehtuurin suunnittelu, komponentin kehitys ja testaus. Datan visualisoinnissa käytettiin Matplotlib-kirjastoa. Tehdyn komponentin oli tarkoitus toimia rinnan omassa prosessissa Nokian testiympäristön kanssa. Valmis työ tulee olla helposti otettavissa käyttöön, testiympäristöstä riippumatta.

Tuloksena saatiin toimiva komponentti Nokian testiautomaatioon, joka voidaan jatkossa ottaa käyttöön eri testimalleissa. Plotteri toimii omassa prosessissaan, joten se on nopea eikä viivytä testiautomaation toimintaa. Asiakkaat käyttävät plotteria mittapisteiden seurantaan ja viallisten mittapisteiden ja radioiden tunnistamiseen.

Jatkossa samankaltainen komponentti haluttaisiin myös Nokian web-pohjaisen testisuunnittelu-työkalun puolelle. Tässä käytetyt kirjastot tukevat myös niihin käyviä osia.

---

Asiasanat: Nokia, Python, ohjelmointi, moniprosessi, matplotlib, datan visualisointi

## ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology,
Option of Software Development

---

Author(s): Lasse Kangas
Title of thesis: Real-time data plotting with Python
Supervisor(s): Manne Hannula
Term and year when the thesis was submitted: Autumn 2023 Number of pages: 28

---

This project was requested by Nokia Mobile Networks at Oulu. Nokia is one of the largest 5G-radio technology companies in the world. Nokia Mobile networks Oulu's site is focused on developing and testing the radios.

The goal of this project is to create a real-time data plotter feature for the Nokia 5G-test automation platform. Plotter source code was written in Python and using the matplotlib-library to create and map the measured points to canvas. Feature is running in its own separate process alongside with the test automation's main process.

The result was working component for Nokia's test automation. This feature is running in its own process and the graph easy fast and easy to follow. This feature is helping customers to identify problems with the tested units.

This feature is planned to be implemented also in the Nokia's web-based test planner tool.

---

Keywords: Nokia, Python, multiprocessing, matplotlib, real-time graph, data plotting

# SISÄLLYS

SANASTO ..... 6
1 JOHDANTO ..... 7
2 PYTHON JA OHJELMISTOKEHITYS ..... 8
2.1 Python-moniprosessi ..... 8
2.2 Matplotlib ..... 9
2.3 Microsoft Visual Studio ..... 10
2.4 Gitlab ..... 10
3 TYÖN TOTEUTUS ..... 11
3.1 Suunnittelu ..... 11
3.2 Plotly kuvaajien testaus ..... 11
3.3 Matplotlib pyplot testaus ..... 12
3.4 Python -moniprosessi ja plotter-luokka ..... 13
3.5 Ensimmäinen toteutus ja HW-testaus ..... 15
3.6 Asiakaspalaute ja toinen versio ..... 16
4 VALMIS KOMPONENTTI ..... 18
4.1 Plotterin käyttöönotto ..... 18
4.2 Datan vienti testiautomaatiolta aliprosessille ..... 21
4.3 Datan hahmottaminen ..... 21
4.4 Plotter-prosessin ja kuvaajien sulkeminen ..... 25
5 JATKOKEHITYS ..... 26
6 YHTEENVETO ..... 27
LÄHTEET ..... 28

## SANASTO

| | |
|---|---|
| Azimuth, elevation | Testiautomaatiossa käytettävien kulmien ja leikkausten nimet. Korvaavat termit ovat horizontal ja vertical |
| Downlink, Tx | Lähettävä |
| DUT | Testattava tuote tai radio. Lyhenne englannista ”Device Under Testing” |
| Git | Versionhallintatyökalu |
| Gitlab | Web-pohjainen projektinhallintatyökalu. |
| HW-testaus | Hardware-testaus, eli oikealla radiolla ja automaatiolla suorittaminen. |
| IDE | Integrated Development Environment. Koodin kirjoittamiseen ja suorittamiseen tarkoitettu ohjelma. |
| Jupyter | Verkkopohjainen interaktiivinen notebook-alusta. |
| Matplotlib | Python kirjasto datan visualisointiin |
| Metodi, funktio | Koodiin kirjoitettu toiminto, jota voidaan kutsua. |
| Python | Ohjelmointikieli |
| Unit test | Yksikkötesti |
| Uplink, Rx | Vastaanottava |

# 1 JOHDANTO

Opinnäytetyön tilaajana oli Nokia Mobile Networks. Oulun Nokian konttori Home of radio perustettiin vuonna 1991 ja se keskittyy radio- ja telekommunikaatiolaitteiden kehittämiseen ja tuottamiseen. Nokian radioita on käytössä ympäri maailmaa. Nykyinen radioiden tuotto- ja kehitys keskittyy 5G-radioihin (8).

Nokia Bell Labs on Oulussa toimiva tietoliikenteen kehityksen ja tuotannon keskus, jossa henkilöstömäärä on noin 2800. Nokia on pitkään ollut 5G-radioiden kehittämisen ja tuotannon edelläkävijä maailmanlaajuisesti. Kehitys ensimmäisistä prototyypeistä valmiiseen radioon tapahtuu Oulun tehtaalla. Valmiit prototyypit ja tuotteet testataan Nokian omalla testiautomaatiolla (8).

Nokia on yksi maailman 5G-radiotekniikan edelläkävijöistä. Nokian radioita on käytössä ympäri maailmaa. Tällä hetkellä Nokialla on yli 50 eri 5G-radiomallia, joita kehitetään ja testataan Oulussa.

Projektin idea sai alkunsa radiotestaajien tarpeesta saada suoraa dataa testattavasta tuotteesta. Kun testattava tuote eli DUT (Device Under Testing) on saatu koottua fyysiseksi tuotteeksi, se testataan Nokian testiautomaatiolla. Testiautomaatiot voivat kestää jopa useamman tunnin, jonka jälkeen tulokset kirjataan Nokian testisuunnittelutyökaluun. Reaaliaikaisessa mittaustulosten seurannassa huomattaisiin mahdolliset virheet ja testiautomaatio voitaisiin keskeyttää. Sen jälkeen mahdolliset virheet korjataan ja käynnistetään testi uudelleen ja saadaan haluttu lopputulos. Työ tehtiin Nokian Test Solutions-tiimin kanssa.

# 2 PYTHON JA OHJELMISTOKEHITYS

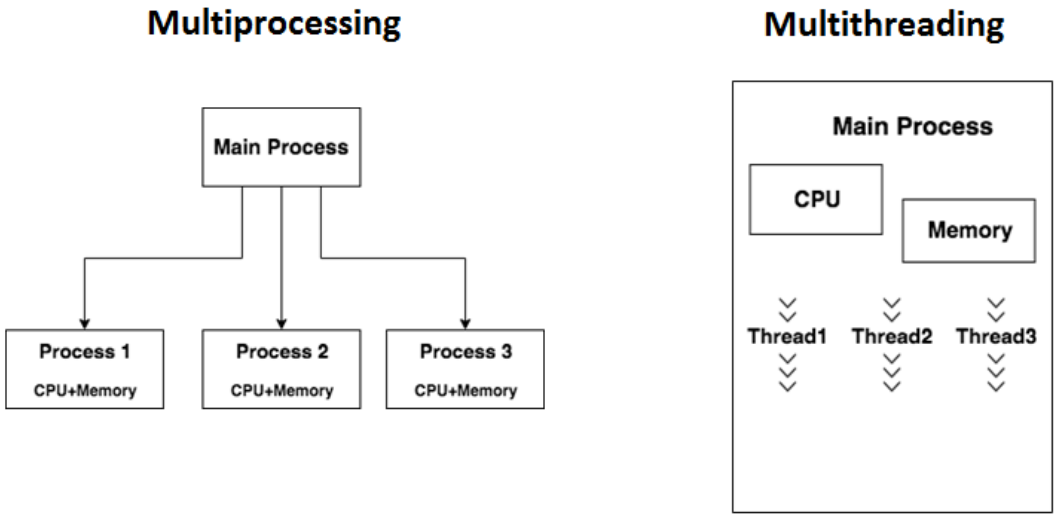
Python-ohjelmointikieli on nykypäivänä lähes käytetyin ohjelmointikieli. Sitä käytetään yleisimmin automatisointiin, datan visualisointiin ja koneoppimisalgoritmien, videopelien, statistiikan ja muiden datapohjaisten asioiden luontiin. Pythonin vahvuudet ovat sen helppolukuisuudessa ja laajassa koodikirjastovalikoimassa. Moni pitää Python-ohjelmointikieltä parhaana vaihtoehtona vasta-alkavalle ohjelmoijalle, kuten myös kompleksisiin ohjelmisto- ja järjestelmäkehityksiin.

Pythonilla voidaan käyttää useita eri koodikirjastoja datan-visualisointiin. Koodikirjastot sisältävät valmiita pohjia ja funktioita, joita ohjelmistokehittäjä voi käyttää omissa koodeissaan. Kirjastot antavat koodille ja komponenteille useita eri toiminnollisuuksia ja käyttötarkoituksia. Tässä työssä käytettiin multiprocessing-kirjastoa moniprosessiin ja matplotlib -moduulia datan visualisointiin.

## 2.1 Python-moniprosessi

Python-moniprosessi (multiprocessing) mahdollistaa useamman prosessin ajoa rinnakkain pääprosessin kanssa. Jokainen prosessi tulee alustaa pääluokan *init*-funktiossa, jossa määritellään jokainen erillinen prosessi. Prosesseille määritellään funktio ja tarvittavat argumentit joita halutaan käyttää uudessa prosessissa (1). Prosessien välillä siirtyvää dataa voidaan lähettää ja vastaanottaa joko queue- tai pipe -metodeita käyttämällä. Nämä metodit periytyvät suoraan multiprocessing-koodikirjastosta. *Queue*-metodilla voidaan lähettää ja vastaanottaa dataa useista eri pisteistä. Pipe-metodilla voi olla vain kaksi pääpistettä, joissa määritellä datan joko yksi- tai kaksisuuntaiseksi. Yksisuuntainen voi ainoastaan joko lähettää tai vastaanottaa dataa, kun taas kaksisuuntaisessa pipe-yhteydessä data voi liikkua molempiin suuntiin pisteiden välillä (5). Tässä työssä käytetään pipe-yhteyttä, sillä se on nopeampi ja riittävä yksisuuntaiseen datan lähettämiseen (2).

Moniajoa Pythonilla voidaan toteuttaa joko omissa prosesseissa tai säikeissä. Molemmat ovat toimivia ratkaisua, mutta molemmilla on omat hyödyt ja rajoitukset. Prosessimoniajon hyötyjä ovat sen oma muistikaista, useamman ytimen hyödyntäminen prosessoreilla, erillisten aliprosessien lopetus ja synkronointi. Prosessimoniajo on käyttöliittymältään hyvin samanlainen kuin monisäie, mutta se käyttää huomattavasti enemmän resursseja toimiakseen (kuva 1).



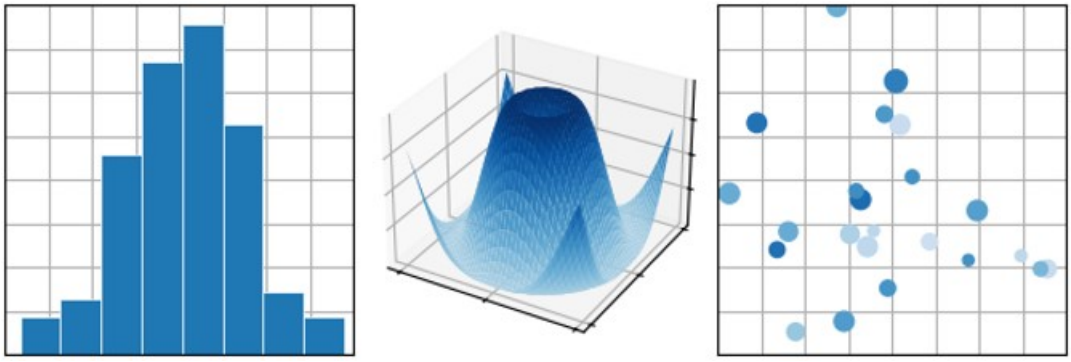

*Kuva 1. Moni- ja säieajon vertailua. (9)*

Monisäieajon hyötyjä on sen keveys, jaettu muisti pääohjelman kanssa, pieni muistinkulutus ja käytettävyys interaktiivisissa ohjelmissa kuten käyttöliittymän suunnittelu. Tässä työssä valittiin käyttöön prosessimoniajo, sillä se sopi parhaiten sen itsenäisen muistiväylän ja työskentelyn myötä. Samalla haluttiin varmistaa, että testiautomaation ja plotterin prosessit toimivat samanaikaisesti, sillä samassa prosessissa olevat säikeet joutuvat välillä odottamaan toisiansa. Multiprosessi toteutuksessa testiautomaatio ei siis hidastu ollenkaan. Myös Nokian radiotestauksessa käytettävät testikoneet ovat riittävän tehokkaita multiprosessoinnin hyödyntämiseen (9).

## 2.2 Matplotlib

Matplotlib on datan ja statistiikan interaktiiviseen hahmottamiseen ja kuvaamiseen perustuva Python -kirjasto. Sen avulla dataa voidaan esittää visuaalisessa muodossa monilla eri kuvaajilla kuten esimerkiksi histogrammilla, 3d- ja hajakuvaajalla. Kirjaston on kehittänyt amerikkalainen neurobiologi John D. Hunter. Matplotlib alun perin kehitettiin auttamaan epilepsiapotilaiden sähkökortikografiatulosten visuaaliseen hahmottamiseen (7).

Kuvaajat voivat olla interaktiivisia, eli pisteitä ja tuloksia voidaan zoomata, panoroida ja päivittää samaan taulukkoon (kuva 2).



*Kuva 2. Matplotlib -kirjastolla luodut histogrammi, 3D ja hajakuvaaja (3).*

### 2.3 Microsoft Visual Studio

Visual Studio on Microsoftin tekemä IDE (Integrated Development Enviroment) -ohjelmisto, joka tarjoaa kattavan ympäristön ohjelmistokehitykseen. Sitä voidaan käyttää koodin kirjoittamiseen ja ajamiseen. VS soveltuu kaikkien koodikielten kirjoittamiseen, mutta toimiakseen ne vaativat lisäosan joka on ladattavissa VS codessa. Tähän kehitysympäristöön on myös mahdollistaa integroida oman lähdekirjaston testejä, joilla koodin toimivuutta testataan. Lisäksi Visual Studioon on mahdollista ladata useita hyödyllisiä työkaluja kuten Live Share, jossa useampi käyttäjä pystyy työskentelemään samassa instanssissa ja reaaliajassa seuraamaan ja opastamaan toisen työskentelyä.

### 2.4 Gitlab

Gitlab on versionhallintatyökalu. Sitä käytetään ohjelmiston eri projektien lähdekoodin säilyttämiseen ja ylläpitoon. Se soveltuu hyvin ketterän kehityksen ylläpitoon. Lähdekoodista voidaan luoda omia haaroja, joissa luodaan uusia ominaisuuksia tai korjataan olevia virheitä koodista. Gitlab tarjoaa myös koodin oikeinkirjoituksen tarkistusta ja yksikkötestien testaamista. Testeillä tarkastetaan koodin kunto ja toimivuus, ennen sen liittämistä lähdekoodiin.

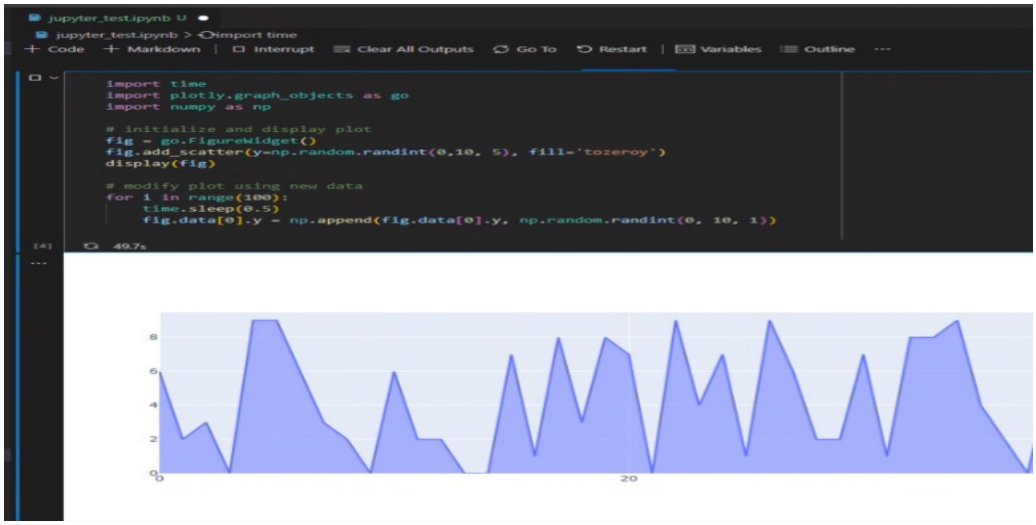
# 3 TYÖN TOTEUTUS

## 3.1 Suunnittelu

Työtä alettiin kartoittamaan ja suunnittelemaan vuoden alussa Nokian Test Solutions-tiimin kanssa. Tarkoituksena oli saada reaaliaikaisten tulosten seuraamiseen helppo ja selkeä plotteri. Vastaavanlainen toteutus oli jo tehtynä Nokian aiempaan testiautomaatioon, mutta se ei toiminut enää nykyisen testiautomaation kanssa. Uusi toteutus haluttiin tehdä siten että pisteiden piirto tapahtuu omassa prosessissa tai säikeessä, jolloin testiautomaatio ei hidastu. Nokian testejä ajetaan niille tarkoitetuilla testikoneilla, jotka ovat yleisimmin Windows-pohjaisia etäkoneita.

## 3.2 Plotly kuvaajien testaus

Nokian testiautomaatiota ajettiin demona käyttäen testattavaa emuloivaa työkalua. Kyseinen työkalu on virtuaalinen palvelin, joka käyttäytyy kuin oikea radio. Kuvaajakomponenttia kehittäessä oli erittäin hyödyllistä päästä testaamaan koodimuutoksia. Aluksi testattiin generoidun pistelistan pohjalta kuvaajaa Jupyterissa käyttäen Plotly -kirjastoa (3). Jupyterin avulla pystytään luomaan ja esittämään Plotlyn luomia kuvaajia, jotka esitetään kuvassa 3 (4).

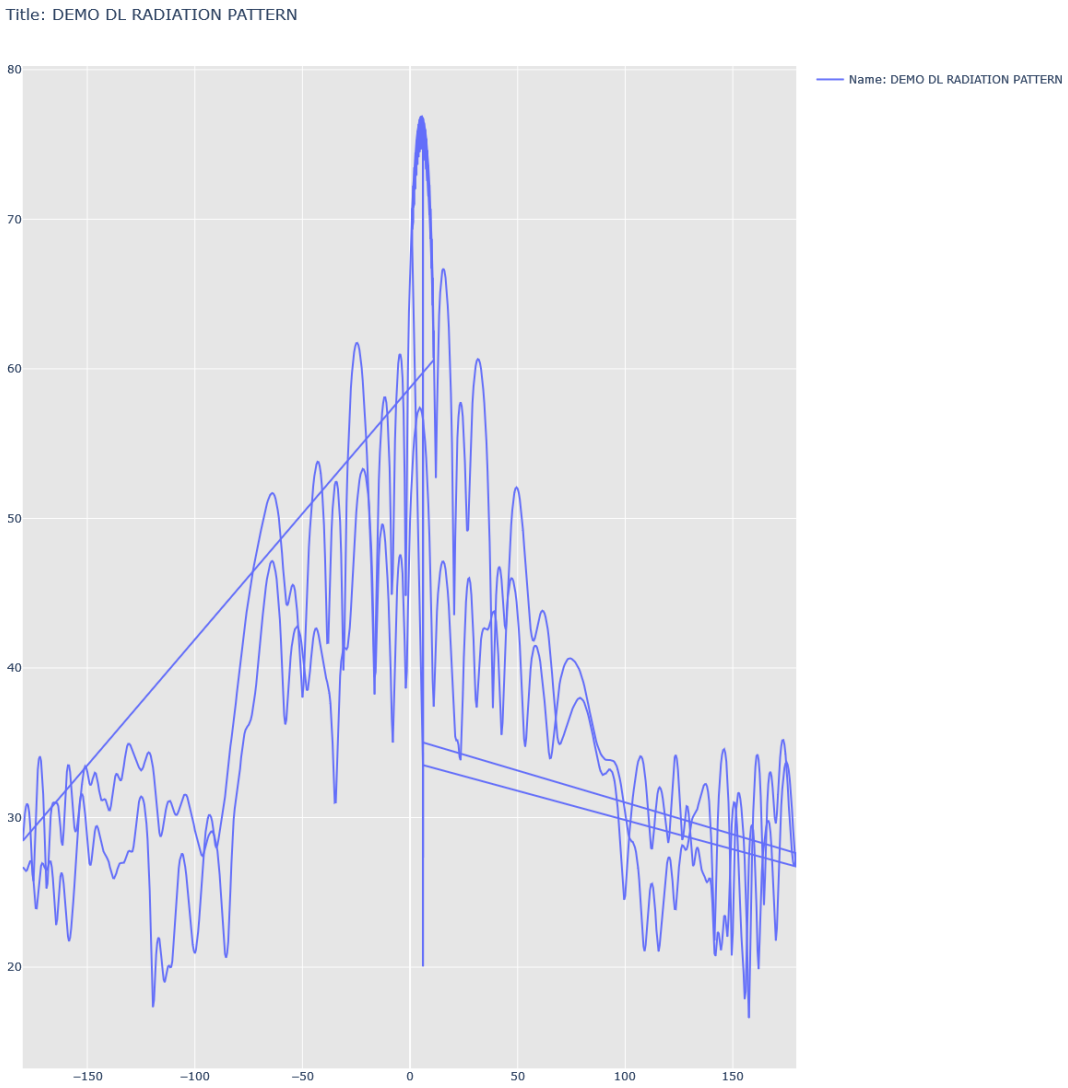



*Kuva 3. Jupyterilla luotu kuvaaja.*

Myöhemmin huomattiin että *Plotly* on hieman monimutkainen toteuttaa omassa prosessissa testiohjelman kanssa.

### 3.3 Matplotlib pyplot testaus

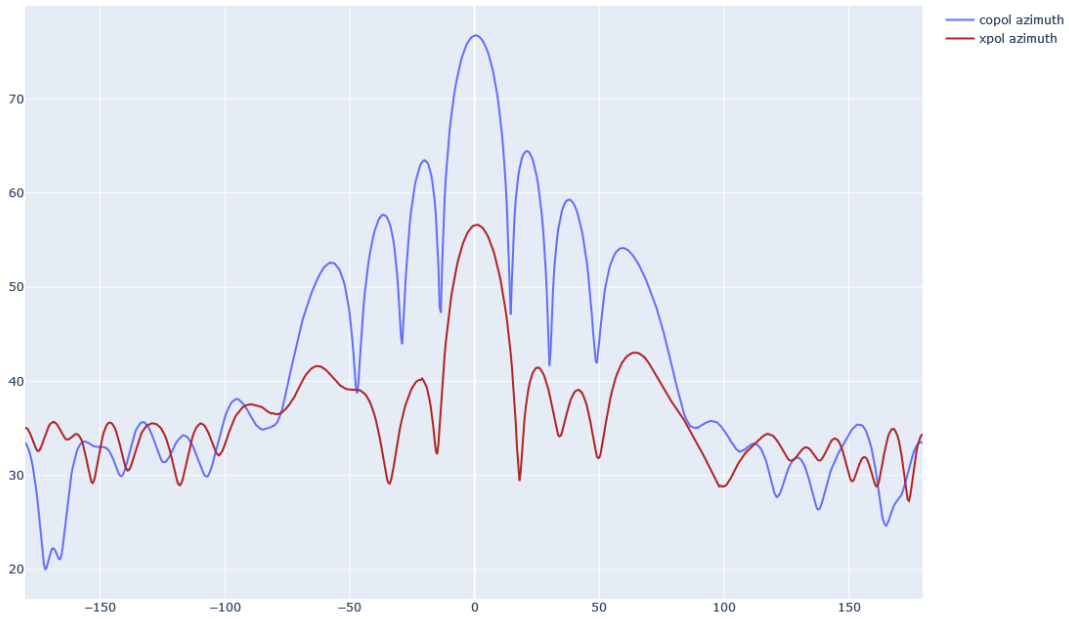
Plotlyn epäonnistuttua otettiin käyttöön Matplotlib -kirjasto. Sen yksinkertaisuus ja keveys osoittautui erittäin toimivaksi ratkaisuksi moniprosessin kanssa. Alkuun testattiin kuvaajan luomista valmiilla mittaustuloksilla (kuva 4).




*Kuva 4. Matplotlib-kuvaaja valmiilla testituloksilla.*

Tulokset saatiin hahmoteltua kuvaajaksi, mutta muutamia virheitä huomattiin, kuten useamman leikkauksen esittäminen samassa kuvaajassa. Kuvaajalle saatiin lisättyä info sekä otsikko, joista ilmenee käyttäjälle mikä kuvaaja on kyseessä.

Koodia kehitettiin ja pian saatiinkin jo paljon selkeämpi kuvaaja, jossa erottuvat tulosten saman kantin eri polariteettimittaukset (kuva 5).




*Kuva 5. Molemmat polariteetit kuvaajassa esitettynä.*

Värien ja infotaulukon myötä saatiin kuvaajalle selkeä tulosten esittäminen. Tähänastisia testejä tehtiin valmiiden pistelistojen kautta.

### 3.4 Python -moniprosessi ja plotter-luokka

Python moniprosessia varten tulee koodiin sisällyttää multiprocessing-moduuli. Tässä tapauksessa multiprocessing lyhennetään ”mp”. Luokassa määritetään pipe-yhteyden vastaanottava ja lähettävä piste (pipe.recv ja pipe.send). Aliprosessia käynnistäessä tulee myös määritellä tarvittavat argumentit, tässä tapauksessa argumentiksi annetaan pipe-yhteyden vastaanottava piste. Aliprosessille voidaan myös antaa oma PID tarvittaessa. Tässä työssä aliprosessi on daemon, eli se sulkee itsensä kun pääohjelma on valmis. Lopuksi aliprosessi käynnistetään komennolla start() (kuva 6) (6).

```python
class LocalPlotter(BasePlotter):
    def __init__(self) -> None:
        """
        While PlotterConfig is True: Initialize plotter as a parallel process running on its own window.
        Uses Pipe-connection to send data from main process to plotter.

        plot_process.daemon = True => Plotter closes graph window when TG-One is finished.
        """
        super().__init__()
        _logger.info("Plotter enabled")
        _logger.info("Plotter: Offset not calculated to points")
        self.start = datetime.now()
        self.start = self.start.strftime("%Y-%m-%d %H:%M:%S,%f")[:-3]

        self.pipe_recv, self.pipe_send = mp.Pipe()
        self.plot_process = mp.Process(target=self.plot, args=(self.pipe_recv,), name='PlotterProcess')
        self.plot_process.daemon = True
        self.plot_process.start()
```

*Kuva 6. Plotter luokka.*

Kun plotter-aliprosessi on aloitettu, testiautomaatio eli pääprosessi aloittaa radion liikuttamisen ja mittaamisen haluttujen mittapisteiden välille. Aina kun saavutaan haluttuun mittapisteeseen se mitataan ja pääprosessi lähettää mittaustuloksen plotterille aliprosessiin käyttäen pipe-yhteyttä.

```python
def plot(self, pipe: Connection) -> None:
    """
    Uses pipe connection to recieve x and y data from the main process.
    Updates x and y arrays with recieved data points and plots them to graph.
    Plots one cut per window.
    Interactive graph windows will stay open until user closes them.

    Args:
        pipe (Connection): Pipe connection to recieve data from send_to_plotter function.
    """
    _logger.info("Plotter started")
    plt.ion()
    _, ax = plt.subplots()
    x_data, y_data = [], []

    while True:
        if pipe.poll():
            data = pipe.recv()
            if data == 'stop':
                break

            x_value, y_value = data

            x_data.append(x_value)
            y_data.append(y_value)

            ax.clear()
            ax.plot(x_data, y_data)
            plt.get_current_fig_manager().set_window_title(f"Started: {self.start}")
            plt.title(self.title)
            plt.xlabel(self.x_label)
            plt.ylabel(self.y_label)
            plt.pause(0.01)
    plt.show(block=True)
```

*Kuva 7. Plot-funktio jossa mittaustulokset esitetään kuvaajassa.*

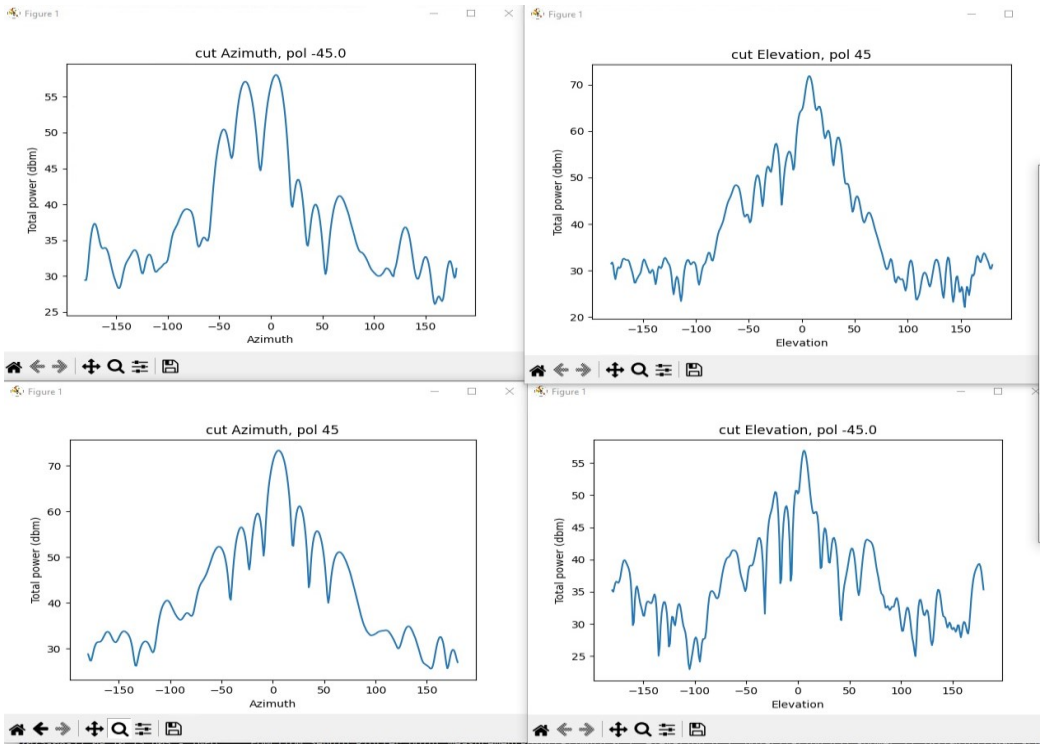
Plot -funktio jaottelee vastaanotetun data x- ja y-akselien datalistoihin ja tämän jälkeen tulokset esitetään kuvaajassa (kuva 7). Jokaiselle mittausleikkaukselle tehdään aina oma aliprosessi ja kuvaaja.

### 3.5 Ensimmäinen toteutus ja HW-testaus

Kun kuvaajan perustoiminnallisuus saatiin toimimaan, lähdettiin koodia testaamaan oikean laitteiston kanssa. Nokian 5G-radioiden testiautomaatio koostuu kolmesta osasta.

- **Testienmallien suunnittelutyökalu**, selainpohjainen palvelin, josta testejä voidaan suunnitella ja laittaa jonoon testaamista varten.
- **DUTin ohjauspalvelu**, testattavan radion palvelintuki.
- **Testausautomaatio**, joka käyttää testien suunnittelutyökalua ja DUTin ohjauspalvelua.

Halutut testit laitetaan jonoon testien suunnittelutyökalusta ja DUTin palvelin päälle etäkoneelta. Tässä toteutuksessa haluttiin aluksi saada toimiva kuvaajanpiirto omassa prosessissa. Jokainen leikkaustulos piirrettiin kuvaajalle ja omassa prosessissaan. Aina kun aloitetaan uusi mittaus niin testiautomaatio luo myös uuden prosessin. Ensimmäisten läpimenneiden testien jälkeen otettiin kuvaajanpiirto käyttöön ja saatiin toimivia ja oikeita tuloksia, jotka esitetään kuvassa 8.

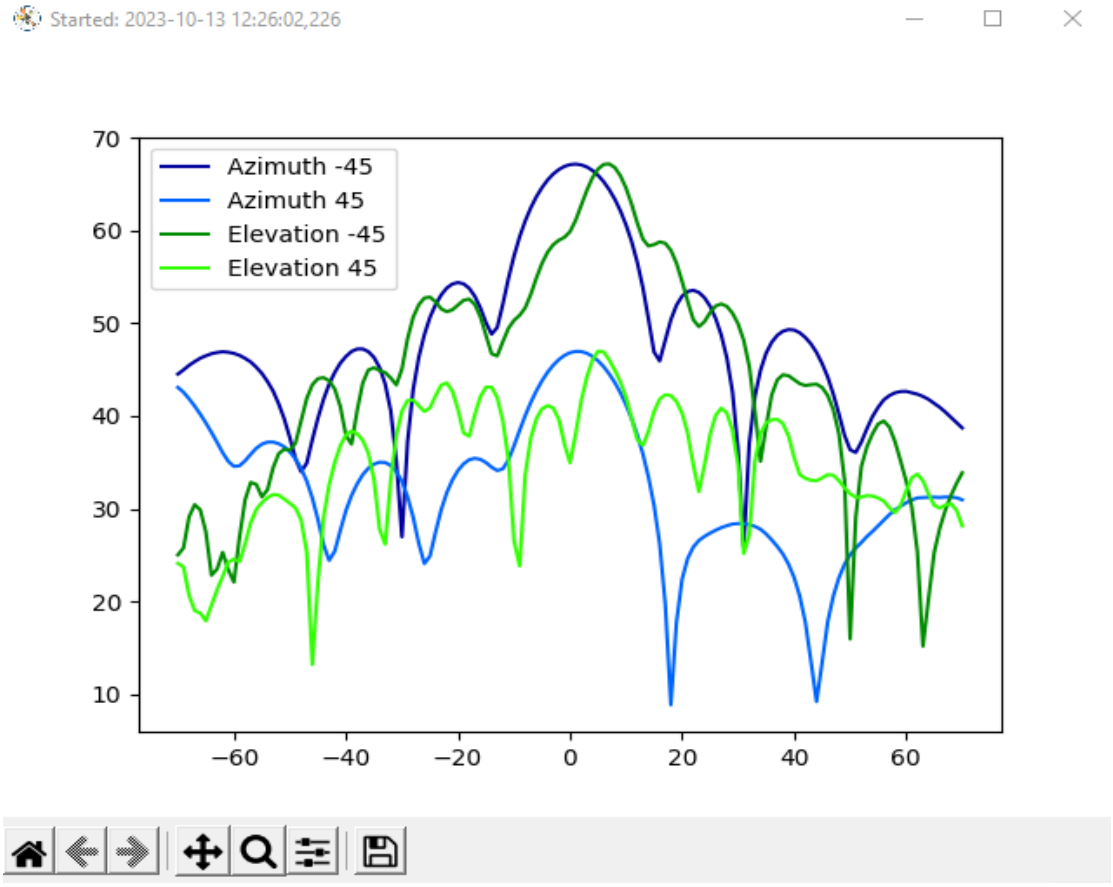



*Kuva 8. Ensimmäisen plotteriversion testituloksia.*

### 3.6 Asiakaspalaute ja toinen versio

Saatuamme toimivan plotteri, halusimme sen asiakkaille käyttöön ja kuunnella heidän mielipiteitään ja jatkokehitysideoita. Asiakkaat halusivat kaikki mitattavat leikkaukset näkymaan samassa kuvaajassa, jolloin mittatuloksia on helpompi vertailla keskenään. Myös värejä ja selkeyttä toivottiin, sillä kun kaikki neljä leikkausta ovat samassa kuvaajassa, olisi sen lukeminen hankalaa. Plotterin toiminnallisuuteen ja sen nopeuteen oltiin tyytyväisiä.

Jatkokehitys ja plotterin toiminnollisuuksien muokkaaminen oli heti huomattavasti nopeampaa toteuttaa ja testata kun koodipohja on tuttu ja vakaa. Aluksi lähdettiin siirtämään kaikkien mittapisteiden piirtoa samaan kuvaajaan sekä muokattiin aliprosessi daemoniksi, jottei se estä testiautomaation päättymistä. Koska kaikki leikkaukset ja mittapisteet esitetään samassa prosessissa ja kuvaajassa, tuli värejä muuttaa, jotta kuvaajan luku olisi selkeämpää. Värit haettiin Nokian uudesta väripaletista. Lisäksi kuvaajan otsikko poistettiin ja tilalle tehtiin infoikkuna kuvaajan vasempaan yläreunaan (kuva 9).




*Kuva 9. Toinen versio jossa kaikki leikkaukset ja mittapisteet ovat samassa kuvaajassa.*

Lisäksi plotterille tehtiin ns. perusluokka BasePlotter. Tämä luokka perii abstraktin luokkaperiaatteen ja sen funktiot ovat abstrakteja metodeja (kuva 10). Abstraktimetodit ovat funktioita, jotka vaaditaan uutta plotteriluokkaa luodessa. Tämä menettelyä käytetään kun halutaan laatia yleiskäyttöön tarkoitettu luokka, jonka funktiot kirjoitetaan uudelleen käytettävissä aliluokissa.

```python
class BasePlotter(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def send_to_plotter(*args) -> None:
        pass

    @abstractmethod
    def plotter_stop(self) -> None:
        pass

    @abstractmethod
    def cut_done(*args) -> None:
        pass

    @abstractmethod
    def terminate_plotter(self) -> None:
        pass

    @abstractmethod
    def plot(point, cut_name, power):
        raise NotImplementedError("Plot not implemented")
```

*Kuva 10. BasePlotter luokka joka perii AbsctractClass -kirjaston.*

# 4 VALMIS KOMPONENTTI

Vaikka aiempien toteutuksien toiminnallisuus ja muut olivat suurimmilta osin toimivia, haluttiin ulkonäköön ja käytännön toiminnollisuuksiin selkeyttä. Risti- ja myötäpolariteettileikkaukset haluttiin omiin kuvaajiin samassa prosessissa. Täten säästetään testauskoneen resursseja. Tulevaisuudessa tarve on myös useamman carrier-tehon mittapisteiden visualisointi kuvaajalle, joten näitä muutoksia ennakointiin plotterin kolmannessa ja viimeisessä versiossa, joka kuuluu tämän projektin sisään.

Kuten aiemmissa plotterin versioissa oli jatkokehityksessä huomattavasti helpompi lähteä tekemään ja havainnollistamaan muutokset lähdekoodissa. Myös useita funktioita kuten mittapisteiden leikkausten nimien tarkistus ja muuntaminen siirrettiin plotter.py tiedoston sisälle. Täten kaikki plotterin tarvittava toiminnallisuus löytyy jatkossakin plotterin sisältä, kun taas ennen se oli testien lähdekoodissa. Lisäksi plotterin logiikkaa mietittiin hieman uudelleen, sillä ongelmia ilmaantui kun testijonossa oli useita testimalleja, mutta vain osa niistä oli tuettuna plotterilla.

## 4.1 Plotterin käyttöönotto

Alkuun tehtiin plotter.py tiedoston sisään oma PlotterHandler -luokka. Tämä luokka lukee käyttäjän täyttämän konfigufointitiedoston, josta löytyy optio käyttää plotteria testiajon aikana. Mikäli config tiedostossa on plotterille asetettu boolean-arvo ”True”, otetaan plotteri käyttöön testiautomaation testicasen pohjustuksessa käyttöön ja luodaan plotterin oma aliprosessi. Tämä aliprosessi myöhemmin tallennetaan PlotterHandler-luokan muuttujaan plotter, josta se saadaan takaisin käyttöön useamman polariteettileikkausten välissä. Näin ei tarvitse luoda uutta plotter-oliota jokaisella mittausleikkauksella (kuva 11).

```
class PlotterHandler:
    """
    enabled (bool): Plotter is enabled/disabled
    first_run (bool): Testcase first cut
    plotter (Object): plotter instance
    x_label (str): name of the cut
    """
    enabled = False
    first_run = True
    plotter = None
    x_label = None
```

*Kuva 11. PlotterHander luokka.*

Jos plotter on otettu käyttöön ja first_run niin luodaan uusi plotter-olio ja set tallennetaan plotter-muuttujaan.

Mikäli kuvaajaa ei haluta käyttöön, luodaan DummyPlotter-luokasta plotter-olio, joka palauttaa tyhjiä käskyjä taikka ei mitään. Tällöin testiautomaatio jatkaa normaalisti ilman kuvaajaa (kuva 12).

```
class DummyPlotter(LocalPlotter):
    def __init__(self) -> None:
        """DummyPlotter is used when PlotterConfig.enable = false.
            This prevents errors when plotter is disabled.
        """
        pass

    def send_to_plotter(*args) -> None:
        pass

    def plotter_stop(self) -> None:
        pass

    def cut_done(*args) -> None:
        pass

    def post_run(self) -> None:
        pass

    def terminate_plotter(self) -> None:
        pass

    def plot(self) -> None:
        pass
```

*Kuva 12. DummyPlotter luokkaa käytetään kun plotteria ei käytetä.*

```python
@classmethod
def get_plotter(self, cut_name: str, swap: bool) -> plotter:
    """
    Check station config if plotter is enabled/disabled.
    LocalPlotter instance is created when plotter is enabled.
    DummyPlotter instance is created when plotter is disabled.

    Args:
        cut_name (str): Cut name
        swap (bool): UL cases when true: Elevation becomes azimuth and vice versa.

    Returns:
        Object: Plotter
    """
    if PlotterHandler.enabled and PlotterHandler.first_run:
        PlotterHandler.plotter = LocalPlotter()
        PlotterHandler.first_run = False
        PlotterHandler.x_label = self.create_plotter_title_labels(cut_name, swap)
        return PlotterHandler.plotter
    elif PlotterHandler.enabled and not PlotterHandler.first_run:
        PlotterHandler.x_label = self.create_plotter_title_labels(cut_name, swap)
        return PlotterHandler.plotter
    else:
        _logger.info("Plotter disabled")
        PlotterHandler.plotter = DummyPlotter()
        return PlotterHandler.plotter
```

*Kuva 13. Get_plotter funktio, jota testiautomaatio kutsuu ennen mittapistesilmukkaa.*

LocalPlotter-luokka, jossa plotter-olio luodaan ja alustetaan sekä käynnistetään aliprosessi, pysyi hyvinkin samanlaisena kuten aiemmissakin versioissa.

## 4.2 Datan vienti testiautomaatiolta aliprosessille

Testiautomaation aloittaessa mittapistesilmukan on plotter-olio luotuna ja sille alustettu pipe-yhteys, jotta mittatuloksia saadaan lähetettyä aliprosessille. Tätä funktiota kutsutaan send_to_plotter ja se on pääprosessin käyttämä.

```
def send_to_plotter(self, x_value: float, y_value: float, polarization: int) -> None:
    """Send x and y data to plotter process via pipe-connection.

    Args:
        x_value (float): Value for x-axis
        y_value (float): Value for y-axis
        polarization (int): Polarity of current cut
    """
    if PlotterHandler.enabled:
        self.pipe_send.send((x_value, y_value, PlotterHandler.x_label, polarization))
```

*Kuva 14. Send_to_plotter funktio jolla on pipe-yhteyden lähettävä pääpiste.*

## 4.3 Datan hahmottaminen

Kuvaajanpiirtämisestä vastaava funktio sai paljon lisäystä viimeiseen versioon. Kuvaajia luodaan samaan ikkunaan kaksi ja niillä on yhteinen tehonmittausarvo keskenään komennolla plt.subplots*(2, sharey=True).* Värejä sekä kuvaajan viivoja muokattiin, jotta niistä erottuvat helpommin polariteettileikkaukset. Lisäksi tehtiin erilliset datalistat molemmille mittapisteleikkauksien kuvaajille. Nyt ylempi kuvaaja on azimuth leikkauksen mittapisteitä varten ja alempi elevation pisteitä varten. Ristipolariteettien mittapisteet esitetään katkoviivalla ja myötäpolariteetin normaalilla viivalla (kuva 15).

```python
def plot(self, pipe: Connection) -> None:
    """
    Uses pipe connection to recieve x and y data from the main process.
    Updates x and y arrays with recieved data points and plots them to graph.
    Azimuth and elevation cuts in separate windows.
    Window is closed when the tc is finished or starting another queued tc.

    Args:
        pipe (Connection): Pipe connection to recieve data from send_to_plotter function.
    """
    _logger.info("Plotter started")
    plt.ion()
    _, ax = plt.subplots(2, sharey=True)
    new_cut = True
    new_azimuth = True
    new_elevation = True
    az_x_lists = [[]]
    az_y_lists = [[]]
    el_x_lists = [[]]
    el_y_lists = [[]]
    az_label_list = []
    el_label_list = []
    az_colour = ['#00009f', '#0064ff', '#00009f', '#0064ff']
    el_colour = ['#d17f00', '#ffb138', '#d17f00', '#ffb138']
    line_list = []
```

*Kuva 15. Omassa aliprosessissaan toimiva plot-funktion alustus.*

Aliprosessin plot-funktiossa käytetään while-silmukkaa mittapistearvojen vastaanottamiseen. Pipe-yhteydellä vastaanotetut arvot jaotellaan omiin listoihin. Silmukkaan on myös asetettu ehdot komennoille aliprosessin sulkemiseen ja mittaleikkauksen päättymiseen (kuva 16).

```
while True:
    if pipe.poll():
        data = pipe.recv()
        if data == 'done':
            az_x_lists.append([])
            az_y_lists.append([])
            el_x_lists.append([])
            el_y_lists.append([])
            new_cut = True
            continue
        if data == 'stop':
            break

        x_value, y_value, x_label, polarization = data

        if x_label == 'Azimuth':
            az_x_lists[-1].append(x_value)
            az_y_lists[-1].append(y_value)
            if new_cut:
                if new_azimuth:
                    az_label_list.append(x_label + ' co-pol')
                    line_list.append('-')
                    new_azimuth = False
                else:
                    az_label_list.append(x_label + ' x-pol')
                    line_list.append('--')
                new_cut = False
```

*Kuva 16. While loop, jossa pipe-yhteydellä vastaanotetaan pääprosessilta data lisätään ne oikeisiin listoihin.*

Kuvassa näkyy kuinka saatu data tarkastellaan ja mikäli kyseinen leikkaus on azimuth, lisätään x- ja y-arvot niille ennalta määrättyihin listoihin. Sama toimenpide tehdään myös elevation leikkauksesta saaduille mittaustuloksille.

Aina kun mittaleikkaus saadaan valmiiksi, lähetetään kyselysilmukkaan pipe-yhteydellä viesti ”done”, jolloin valmiit x- ja y-data listoihin lisätään uudet listat, joihin seuraavan leikkauksen mittapisteet lisätään.

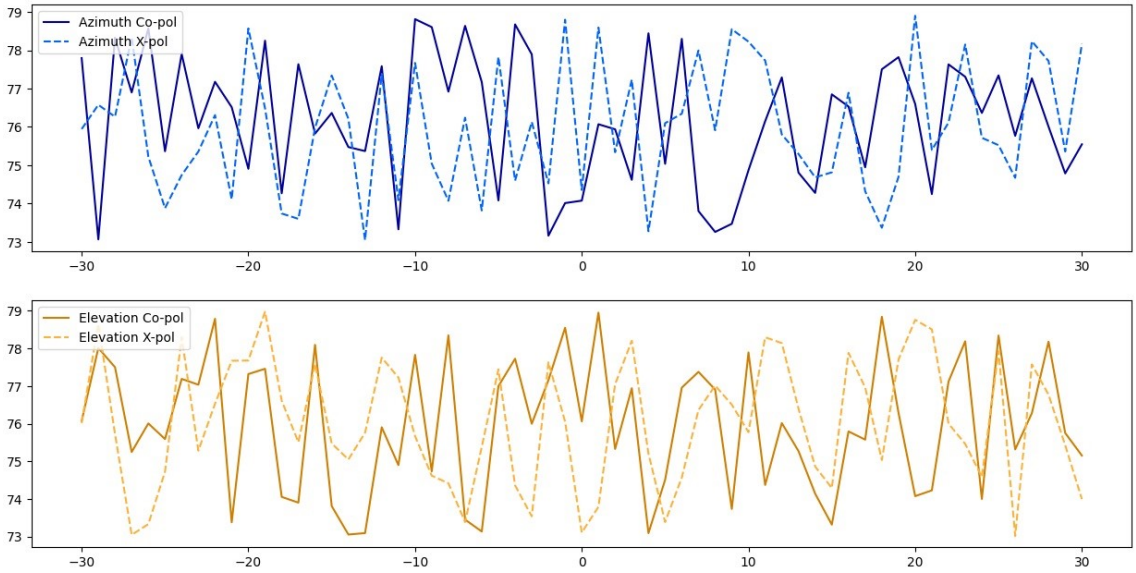
Kun data on lajiteltuna oikeisiin listoihin ne esitetään kuvaajissa. Molemmille kuvaajille on omat listat x-, y-, viivojen tyylittely- ja värilistat. Ne käydään läpi samassa *for*-silmukoissa kuvassa esitetyllä tavalla (kuva 17).

```
        ax[0].clear()
        ax[1].clear()
        for idx in range(len(az_x_lists)):
            ax[0].plot(az_x_lists[idx],
                       az_y_lists[idx],
                       linestyle=line_list[idx],
                       color=az_colour[idx])
            ax[1].plot(el_x_lists[idx],
                       el_y_lists[idx],
                       linestyle=line_list[idx],
                       color=el_colour[idx])

        ax[0].legend(az_label_list, loc="upper left")
        ax[1].legend(el_label_list, loc="upper left")
        plt.get_current_fig_manager().set_window_title(f"Started: {self.start}")
        plt.pause(0.01)
plt.show(block=True)
```

*Kuva 17. For-silmukka jossa molemmat kuvaajat päivitetään uusilla listoilla*

Kun päivitetyt listat on kuvaajassa käytetään vielä molemmissa infolaatikkoa, joista asiakkaalle näkyy mitä kyseinenkin käyrä kuvaajassa on (kuva 18).




*Kuva 18. Plotterin luomat kuvaajat joissa esitettynä molemmat polariteetit.*

### 4.4 Plotter-prosessin ja kuvaajien sulkeminen

Kun kaikki mitattavat pisteet on mitattu ja niiden data hahmoteltuna kuvaajiin, taikka mitattavat pisteet eivät kuulu azimuth ja elevation leikkauksiin, esim. peak_scan, lähetetään plotter-prosessille viesti pipe-yhteydellä ”stop”, joka pysäyttää kyselysilmukan. Kuvaajat jäävät käyttäjälle näkyviin ja odottavat pääprosessin eli testiautomaation sulkemista. Kaikkien mittausten jälkeen pääprosessi suorittaa resurssien vapauttamisen, mittaustulokset lähetetään Nokia testisuunnittelutyökalulle ja plotter-prosessille suoritetaan sulkemiskäsky terminate(). Samalla alustetaan uusi plotter-olio tyhjäksi (kuva 19).

```
@classmethod
def post_run(self) -> None:
    """Executed after tc is finished.
    """
    if PlotterHandler.enabled:
        try:
            PlotterHandler.plotter.terminate_plotter()
            PlotterHandler.plotter = None
            PlotterHandler.first_run = True
        except AttributeError:
            pass
```

*Kuva 19. Plotterin post_run funktio, joka suoritetaan pääohjelma on valmis.*

# 5 JATKOKEHITYS

Projektin tärkeimmät vaatimukset saatiin toteutettua. Jatkossa plotterille tullaan tekemään tuki myös useamman teholähteen mittaamiseen ja niiden tehoarvojen esittäminen kuvaajissa. Nykyinen toteutus on tehty tukemaan ainoastaan yhdelle lähettävälle ja vastaanottavalle testiautomaatiolle. Koska plotteri on omana tiedostonaan Nokian testausautomaatiossa, on se helppo sisällyttää myös muihin testicaseihin.

Plotterista halutaan myös yleiskäytännöllinen versio, jolla pystyttäisiin havainnollistamaan mitä tahansa dataa antamalla plotterille suoraan x- ja y-akseleiden arvoja.

Plotterin funktiot ovat tällä hetkellä ilman yksikkötestejä, mutta ne tullaan sinne lisäämään, jotta plotterin lähdekoodia on helpompi ylläpitää ja mahdolliset virheet havaitaan ennen kuin uusia toimintoja käytetään oikean raudan kanssa.

Jatkossa samankaltainen komponentti haluttaisiin myös Nokian web-pohjaisen testisuunnittelutyökalun puolelle. Tässä käytetyt kirjastot tukevat myös niihin käyviä osia.

# 6 YHTEENVETO

Projektin tarkoituksena oli luoda komponentti, jolla Nokian testiautomaation käyttäjät voivat seurata mittapisteiden tehoarvoja reaaliajassa. Komponentin tuli olla nopea, selkeästi luettava ja helposti implementoitavissa mahdollisiin uusiin käyttötarpeisiin.

Työ saatiin haluttuun loppupisteeseen onnistuneesti. Reaaliaikaisten pistemittausten tulokset piirtyvät omiin kuvaajiinsa. Plotter-komponentti toimii omassa erillisessä prosessissa eikä hidasta Nokian testiautomaatiota. Versiohallinnassa käytetty GitLab oli hyödyllinen, sillä aina kun komponentin koodia päivitettiin ja lisättiin lähdekoodiin, sen oikeinkirjoitus ja mahdolliset virheet huomattiin testien avulla.

Projektissa tulleet ongelmat saatiin ratkaistua ja koko ominaisuus hiottua siistiksi ja toimivaksi. Moniprosessiajoa ja reaaliaikaisen datan siirtoa varten jouduttiin testaamaan useampaa eri vaihtoehtoja, kuten Pythonin multithreadingia ja queue -funktioita datan siirtämiseen.

Datan hahmottamiseen meni kuitenkin eniten aikaa, sillä suoraan vastaavanlaisia toteutuksia taikka esimerkkejä ei löydetty. Kuvaajien luomiseen, sekä niiden päivittämiseen oli useita eri menetelmiä. Tässä työssä kuitenkin päädyttiin käyttämään multiprosessia ja Matplotlib-kirjastoa. Nykyinen toteutus on helposti implementoitavissa muuallekin Nokian testausautomaatioon.

Komponentin kehityksessä opittiin hyvin paljon suunnittelusta, tuotetestauksesta ja ohjelmistoarkkitehtuurista. Python olio-ohjelmoinnista oppi edelleen uutta abstraktiluokkien ja -metodien myötä. Datan hahmottamisessa ja front-end ohjelmoinnissa oli visuaalisten tulosten saaminen uutta tekijälle.

Lopputuloksena on uudelleenkäytettävä komponentti, josta voidaan muokata tarpeisiinsa ominaisuus testaus- ja kehitysympäristöstä riippuen.

# LÄHTEET


1 Python 2023. multiprocessing — Process-based parallelism. Hakupäivä 20.2.2023. https://docs.python.org/3/library/multiprocessing.html

2 Python 2023. Pipes and Queues. Hakupäivä 27.2.2023. https://docs.python.org/3/library/multiprocessing.html#pipes-and-queues

3 Plotly 2023. Plotly Open Source Graphing Library for Python. Hakupäivä 15.3.2023. https://plotly.com/python/

4 Plotly community 2019. Plot real-time data. Hakupäivä 17.3.2023. https://community.plotly.com/t/plot-real-time-data/31467

5 Brownlee, Jason 2022. Multiprocessing Pipe in Python. Hakupäivä 21.3.2023. https://superfastpython.com/multiprocessing-pipe-in-python/

6 Brownlee, Jason 2023. Python multiprocessing: The Complete Guide. Hakupäivä 18.4.2023. https://superfastpython.com/multiprocessing-in-python/

7 Matplotlib 2023. Matplotlib: Visualization with Python. Hakupäivä 18.4.2023. https://matplotlib.org/

8 Nokia 2023. Oulu, Finland. Hakupäivä 12.6.2023. https://www.bell-labs.com/about/locations/oulu-finland/

9 Uz Zaman, Nouer 2023. Tug of War: Multiprocessing Vs Multithreading. Hakupäivä 11.6.2023. https://medium.com/@noueruzzaman/tug-of-war-multiprocessing-vs-multithreading-55341c1f2103