Ville Ahola

IR-spektrometrisensorin esimerkkiapplikaatio Raspberry Pi -alustalle

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Sähkötekniikan koulutusohjelma

Insinöörityö

13.5.2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Ville Ahola IR-spektrometrisensorin esimerkkiapplikaatio Raspberry Pi -alustalle 43 sivua + 1 liite 13.5.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	sähkötekniikka
Suuntautumisvaihtoehto	elektroniikka
Ohjaajat	Senior R&D Engineer Jussi Mäkynen lehtori Janne Mäntykoski

Tässä insinöörityössä rakennettiin ohjelma infrapuna-alueen mittauksiin tarkoitetun optisen spektrometrisensorin ohjaamiseen. Sensorin toiminta perustuu Fabry-Perot-interferometriin. Työssä käytettiin laitealustana Raspberry Pi -minitietokonetta, johon oli liitetty pieni kosketusnäyttö.

Ohjelman tehtävänä oli lukea sarjaportin välityksellä sensorilta spektri, laskea spektristä kemometrisen mallin avulla näytteen konsentraatio ja näyttää lopputulos näytöllä. Ohjelma rakennettiin Linux-pohjaisen Raspbian-käyttöjärjestelmän päälle. Pääasiallinen projektissa käytetty ohjelmointikieli oli C++, mutta sarjaporttikommunikaatio toteutettiin Linuxin C-kielisillä systeemikutsuilla.

Graafinen ulkoasu toteutettiin Qt-kirjastolla. Käyttöliittymä suunniteltiin Qt Creatorin WYSIWYG-editorissa. Spektri piirrettiin näytölle kolmannen osapuolen kehittämää Qt-moduulia nimeltä QCustomPlot apuna käyttäen.

Tuloksena syntyi ohjelma, jossa on kaksi näkymää: spektrinäkymä ja analyysinäkymä. Spektrinäkymä näyttää anturin mittaaman raakaspektrin ja analyysinäkymä näyttää kemometrisen mallin avulla lasketun konsentraatioarvon. Ohjelman käyttöliittymä on vielä melko keskeneräinen ja vaatii jatkokehitystä.

Avainsanat	Qt, Raspberry Pi, sarjaportti, spektroskopia

Author	Ville Ahola
Title	Example Application for an IR Spectrometer Sensor Using Raspberry Pi
Number of Pages	43 pages + 1 appendix
Date	13 May 2015
Degree	Bachelor of Engineering
Degree Programme	Electrical Engineering
Specialisation option	Electronics
Instructors	Jussi Mäkynen, Senior R&D Engineer Janne Mäntykoski, Senior Lecturer

In this thesis a program was built for controlling an optical spectrometer sensor measuring at the infrared range. The sensor is based on the Fabry-Perot interferometer. The hardware platform utilized in the project was a Raspberry Pi combined with a small touch screen.

The program's task was to read the spectrum measured by the sensor through the serial port, calculate the sample concentration using a chemometric model and display the result on screen. The program was built on top of the Linux based Raspbian operating system. The main programming language used in the project was C++, but the serial port communication was implemented using system calls provided by the operating system, which were in C.

The graphical user interface was implemented using the Qt framework. The user interface was designed in the WYSIWYG editor provided by Qt Creator. The spectrum was drawn on screen using a third-party Qt module called QCustomPlot.

The result was a program that has two views: spectral view and analysis view. The spectral view presents the spectrum measured by the sensor as is, while the analysis view displays the concentration value calculated using a chemometric model. The program’s interface is still incomplete and requires further development.

Keywords	Qt, Raspberry Pi, serial port, spectroscopy

Sisällys

Liite 1. Lähdekoodi

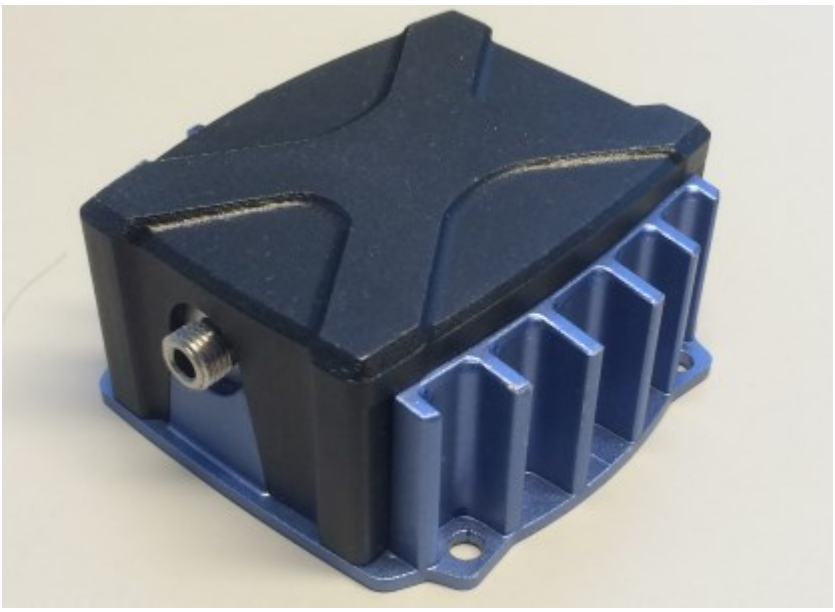
Lyhenteet

ARM	Advanced RISC Machines. Prosessoriarkkitehtuuri.
DBR	Distributed Bragg Reflector. Kerroksittain kasvatettu heijastava pinta.
DCD	Data Carrier Detect. Sarjaportin signaali, joka kertoo, onko yhteys kahden laitteen välillä olemassa.
DSI	Display Serial Interface. MIPI Alliance -järjestön määrittelemä spesifikaatio näyttöliitännälle.
FPI	Fabry-Perot-interferometri. Kahden vastakkaisen, osittain läpäisevän peilipinnan avulla toteutettu interferometri.
FWHM	Full Width Half Maximum. Transmissiopiikin leveys puolivälissä maksimiarvosta.
GPIO	General Purpose Input/Output. Liitäntä, jonka pinnit voi määritellä vapaasti joko sisään- tai ulostulopinneiksi.
GPL	GNU General Public License. Vapaan lähdekoodin ohjelmistolisenssi.
HDMI	High Definition Multimedia Interface. Kuvan ja äänen siirtämiseen tarkoitettu liitäntä.
I/O	Input/Output. Syöte ja ulostulo.
InGaAs	Indium-gallium-arsenidi. Detektorityyppi.
MCT	Elohopea-kadmium-telluridi. Detektorityyppi.
MEMS	Micro Electro Mechanical System. Mikroskooppisen pieni mekaaninen järjestelmä.
MIPI	Mobile Industry Processor Interface. MIPI Alliance -järjestön määrittelemä spesifikaatio mobiililaitteiden liitännille.

MOC	Meta-object compiler. Metaobjektikääntäjä, joka kääntää Qt:n oman syntaksin standardiksi C++-kieleksi.
PbSe	Lyijy-seleeni. Detektorityyppi.
PCA	Principal Component Analysis. Regressioanalyysin menetelmä.
PCR	Principal Component Regression. Regressioanalyysin menetelmä.
POSIX	Portable Operating System Interface. Standardi joka määrittelee rajapinnan kommunikointiin Unix-pohjaisten käyttöjärjestelmien kanssa.
SPI	Serial Peripheral Interface Bus. Synkroninen neljän johtimen sarjaväylä.
USB	Universal Serial Bus. Sarjaväylä.
VTT	Teknologian tutkimuskeskus VTT.

1 Johdanto

Tässä insinöörityössä rakennetaan Raspberry Pi -pohjainen ohjausjärjestelmä spektrometrianturille. Työ tehdään Spectral Engines Oy:lle, joka on VTT:ltä (Teknologian tutkimuskeskus VTT) irroittautunut spin-off-yritys. Yritys valmistaa kämmenelle mahtuvia spektrometriantureita infrapuna-alueelle. Kuvassa 1 on esimerkki anturista.



Kuva 1. Spektrometrianturi

Anturi itsessään ei ole kokonainen mittalaite, vaan pelkästään osa sellaista. Kokonainen spektrometri sisältää anturin lisäksi valonlähteen, kyvetin ja tietokoneen tai mikro-ohjainkortin, joka prosessoi anturin lähettämää dataa ja välittää sitä eteenpäin halutussa muodossa. Antureita on tarkoitus myydä yrityksille, jotka kytkevät siihen edellämainitut osat ja paketoivat kokonaisuuden omaksi tuotteekseen.

Kuvassa 2 esitetään mittalaitteen osat tietokonetta lukuunottamatta. Kuvassa on kaksi mittausjärjestelmää, joista ylemmässä järjestelmässä on läpivirtauskyvetti, jonka läpi voi pumpata näyteainetta tasaisena virtana. Alemmassa järjestelmässä on perinteisempi nestekyvetti.

Kuva 2. Mittausjärjestelmiä

Työssä rakennetaan järjestelmän tietojenkäsittelyosa hyödyntämällä Raspberry Pi –minitietokonetta ja sen Linux-pohjaista Raspbian-käyttöjärjestelmää. Käyttöliittymä rakennetaan alustariippumattomalla Qt-kirjastolla. Lopputulos on suunniteltu lähinnä markkinointitarkoituksiin ja stimuloimaan potentiaalisten asiakkaiden mielikuvitusta.

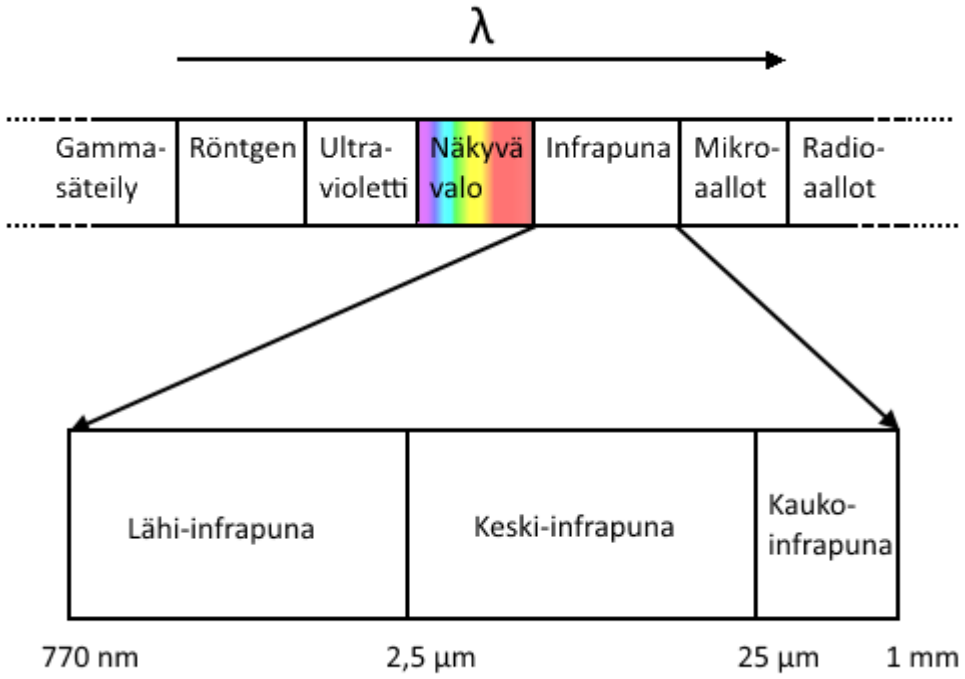
Dokumentissa esitellään työ ja sen tulokset, sekä taustoitetaan anturin toimintaperiaatetta käymällä pintapuolisesti läpi spektroskopian perusteita yleisesti sekä anturiin MEMS (Micro Electro Mechanical System) -tekniikalla rakennetun Fabry-Perot-interferometrin toimintaa erityisesti.

2 Spektroskopia

Spektroskopia perustuu sähkömagneettisen säteilyn ja aineen väliseen vuorovaikutukseen. Eri aineet reagoivat sähkömagneettiseen säteilyyn eri tavalla. Vuorovaikutus on riippuvainen aallonpituudesta, joten kullekin aineelle on muodostettavissa sille tyypillinen spektri. Spektroskopia on siis spektrometriaa. Spektrometrialla tarkoitetaan mate-

riaaleille tyypillisten spektrien mittaamista. Spektrin mittaavaa järjestelmää kutsutaan vuorostaan spektrometriksi.

Sähkömagneettisen säteilyn spektri esitetään kuvassa 3. Erilaiset mittaustekniikat hyödyntävät eri osia spektristä. Spektrin alueet jaetaan yleensä vielä pienempiin osa-alueisiin. Infrapuna-alue jaetaan tyypillisesti lähi-, keski- ja kaukoinfrapunaan. Nimitys kertoo, kuinka kaukana näkyvästä valosta alue on sähkömagneettisen säteilyn spektrissä. Infrapuna-alue kattaa aallonpituudet välillä 0,77 µm – 1 mm.



Kuva 3. Sähkömagneettisen säteilyn spektri

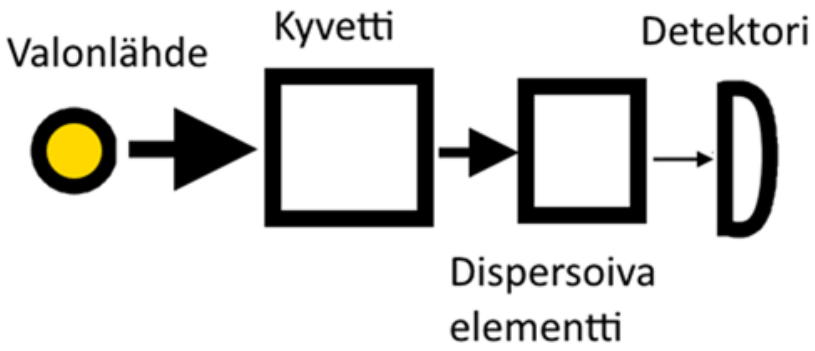
Spektroskopiassa voidaan tarkastella joko emissiota, absorptiota tai sirontaa. Emissiossa atomi tai molekyyli saadaan hetkellisesti vaihtamaan energiatilaa, jolloin tilan palautuessa ylimääräinen energia purkautuu säteilynä. Absorptiossa molekyyli absorboi säteilyä tietyillä aallonpituuksilla rakenteensa takia. Sirontaa taas käytetään esimerkiksi Raman-spektroskopiassa, jossa molekyyliin ammutaan monokromaattista (vain yhtä aallonpituutta sisältävää) valoa, joka siroaa. Osa sironneesta valosta poikkeaa aallonpituudeltaan alkuperäisestä monokromaattisesta valosta. Emissio- ja Raman-mittauksia tehdään lähinnä näkyvän ja ultraviolettivalon alueella, kun taas absorptiota voidaan mitata koko spektrin alueella.

Mittaus voi tapahtua transmittanssimittauksena, jolloin valo kulkee näytteen läpi, tai reflektanssimittauksena, jolloin valo heijastuu näytteestä anturille. Jos mittauksessa käytetty säteily läpäisee näytteen liian heikosti ja detektorin signaalitaso on liian matala, voidaan käyttää reflektanssimittausta.

Spektrometrinen mittaus voi perustua useaan erilaiseen säteilyn ja näytteen väliseen vuorovaikutukseen. Molekyylejä mitattaessa sähkömagneettinen säteily voi aiheuttaa muutoksia molekyylin pyörimisessä, värähtelyssä tai sähköisessä tilassa, tai jopa ionisoida molekyylin. Atomeja mitattaessa ilmiöt rajoittuvat sähköisiin tilamuutoksiin ja ionisoitumiseen. Säteilyn vaikutus riippuu säteilyn energiasta. Tässä työssä käytetty anturi mittaa säteilyä lähi-infrapuna-alueella, jossa mittaus perustuu molekyylin värähtelyyn.

Näytteen olomuoto ja mittausolosuhteet vaikuttavat mittaukseen. Erityisesti pyöriminen heikkenee nesteessä ja käytännössä lakkaa kiinteässä aineessa. Värähtelyn spektriin tuottamat absorptiopiikit myös levenevät nesteessä ja kiinteässä aineessa, mikä heikentää mittauksen resoluutiota. Korkeaa resoluutiota vaativissa mittauksissa näyte pidetään yleensä matalapaineisena kaasuna.

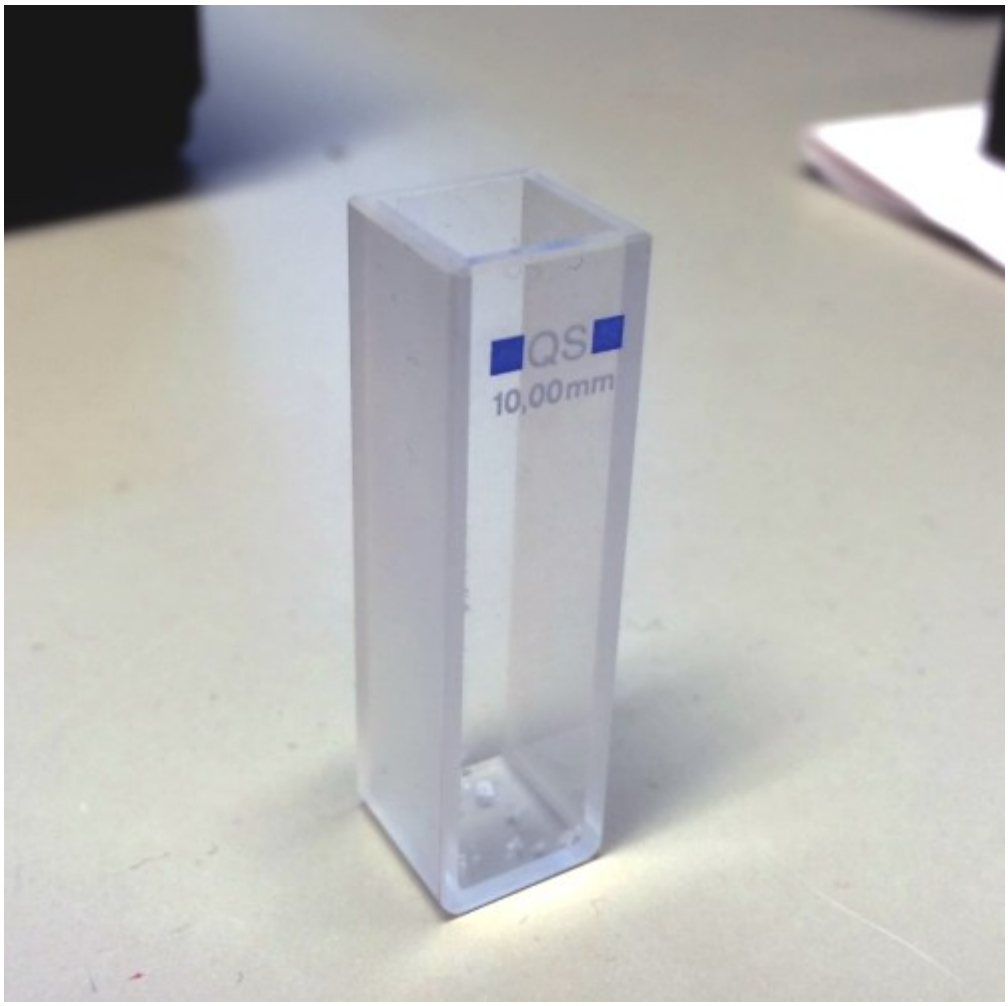
Spektrometri muodostuu tyypillisesti valonlähteestä, kyvetistä, valon spektriksi hajottavasta (dispersoivasta) osasta ja valoherkästä detektorista. Kuten kuvasta 4 selvästi näkyy, valonlähde lähettää laajalla spektrillä valoa, josta osa absorboituu näytteeseen. Näytteen kautta kulkeneesta valosta poimitaan dispersoivalla elementillä haluttu aallonpituus detektorille.



Kuva 4. Spektrometrin rakenne

Valonlähde voi olla esimerkiksi hehkulamppu. Sen lähettämän sähkömagneettisen säteilyn tulisi olla laajakaistaista ja intensiteetti tasaista koko mitattavalla alueella.

Kyvetiksi kutsutaan optisissa mittauksissa astiaa, jossa mitattavaa näytettä pidetään. Siinä on ikkunat, joiden kautta säteily pääsee kulkemaan valonlähteestä näytteen läpi dispersoivalle elementille. Ikkunoiden tulee olla sellaista materiaalia, että ne päästävät valoa mitattavilla aallonpituuksilla. Lisäksi kyvetin tulee olla tarpeeksi kookas, jotta valonsäde kulkee riittävän pitkän matkan mitattavan näytteen sisällä ja absorptiota ehtii tapahtua mittalaitteella havaittava määrä. [1, s. 41−43.] Kuvassa 5 on esimerkki kyvetistä.



Kuva 5. Kyvetti

Detektorin tulee olla herkkä mitattavalle aallonpituusalueelle. Detektorit ovat usein valoherkkiä puolijohteita. Lähi-infrapuna-alueella käytetään tyypillisesti InGaAs (indium-

gallium-arsenidi) -detektoreita ja keski-infrapuna-alueella PbSe (lyijy-seleeni) ja MCT (elohopea-kadmium-telluridi) - detektoreita [2, s. 193].

Valoa dispersoiva osa voi olla esimerkiksi prisma, hila tai interferometri. Järjestelmän anturi perustuu näistä viimeiseen, joten sitä käsitellään tässä työssä hieman enemmän, kun taas kaksi ensimmäistä mainitaan vain ohimennen.

Dispersoivaan osaan liittyy kiinteästi sen erottelukyky. Erottelukyky aallonpituudelle λ määritellään yhtälöllä $R = \frac{\lambda}{\Delta\lambda}$. $\Delta\lambda$ on minimiaallonpituusväli, jolla kaksi aallonpituutta ovat vielä erotettavissa toisistaan. Jos kaksi sädettä, joiden aallonpituudet ovat λ_1 ja λ_2 eivät täytä ehtoa $|\lambda_1 - \lambda_2| \geq \Delta\lambda$, ne näkyvät mittauksessa samana aallonpituutena. [1, s. 43.]

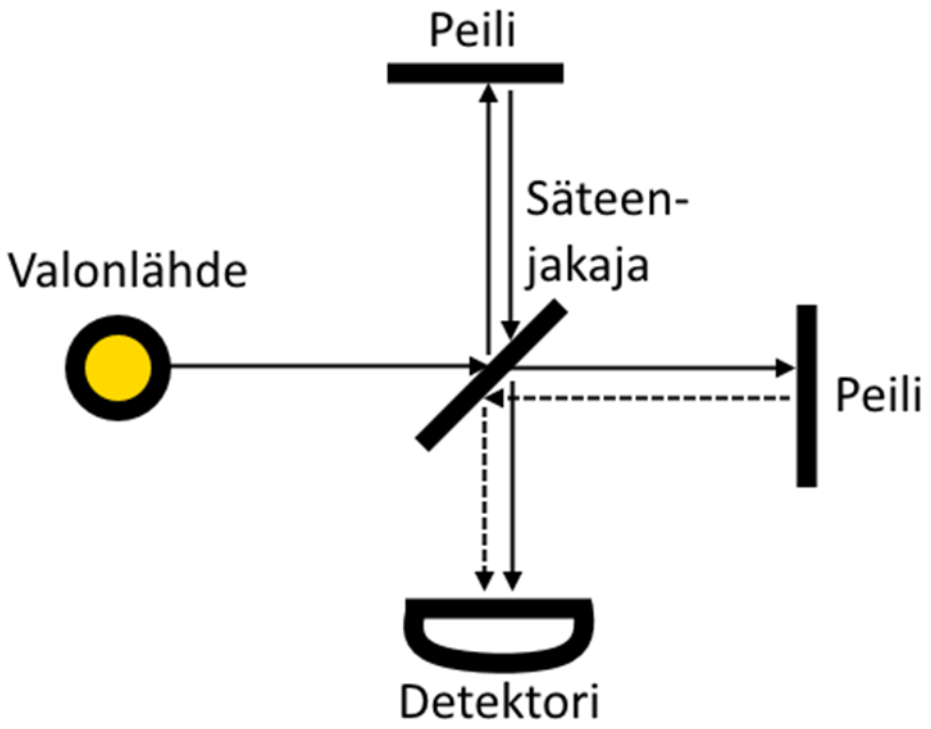
Klassinen esimerkki prismasta on kolmionmuotoinen lasipala, mutta prismoja voidaan valmistaa monenmuotoisina ja eri materiaaleista. Spektroskopiassa prismojen käyttö perustuu prismassa aiheutuvien heijastusten kulmien taajuusriippuvuuteen. Eri taajuiset valonsäteet tulevat prismasta ulos eri kulmissa. [3, s. 187−189.]

Hilan aiheuttama dispersio perustuu diffraktioon, joka tapahtuu valonsäteen kulkiessa reiän läpi. Diffraktiossa osa valonsäteestä osuu esteeseen, jolloin sen amplitudi tai vaihe muuttuu. Muuttunut osa valonsädettä interferoi muuttumattoman osan kanssa. Tarkalleen ottaen diffraktio on siis interferenssiä, mutta lähinnä historiallisista syistä nämä käsitteet pidetään edelleen erillisinä. [3, s. 443.]

Interferometri perustuu nimensä mukaisesti interferenssiin. Interferenssissä kaksi tai useampia sähkömagneettisia aaltoja summautuu siten, että summan intensiteetti on jotain muuta kuin intensiteettien itseisarvojen summa. Jos interferoivia aaltoja on n kappaletta ja aaltojen intensiteettien itseisarvot ovat yhtä suuret, eli jos aallot voivat erota toisistaan vain vaiheensa suhteen, summan intensiteetti vaihtelee välillä [0, n*I], missä I on yksittäisen aallon intensiteetin itseisarvo. [4, s. 54−55.]

Interferometrijärjestelmiä on useita erilaisia. Mainittakoon tässä Michelsonin, Mach-Zehnderin, Sagnacin ja Fabry-Perot'n interferometrit. Näistä viimeinen löytyy järjestelmän anturin sisältä, joten se käsitellään hieman perusteellisemmin omassa osiossaan.

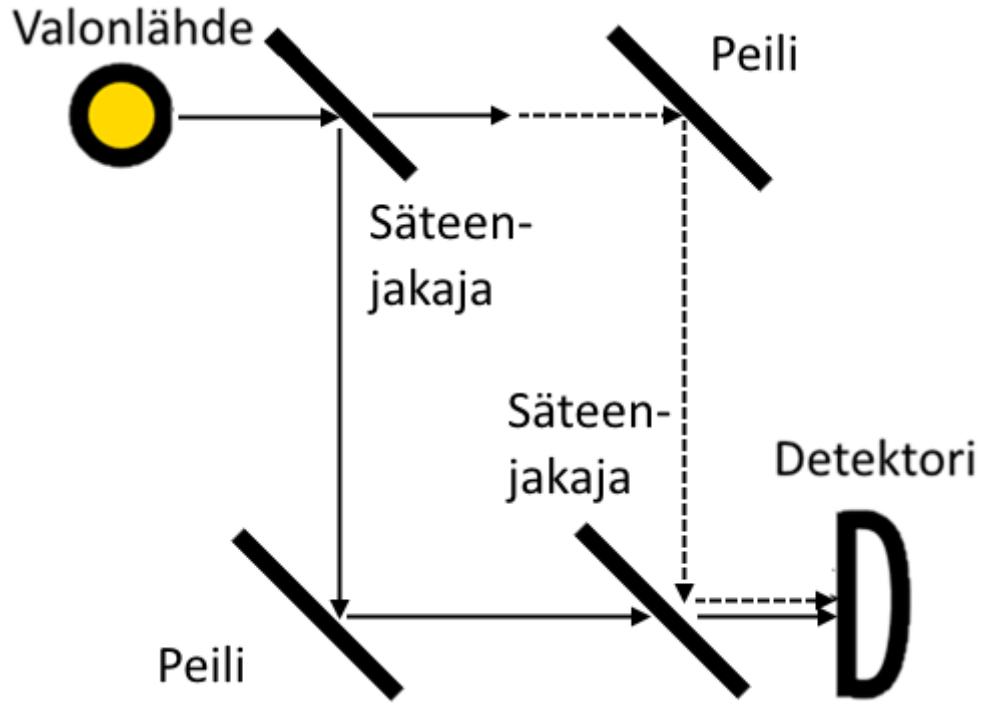
Michelson-interferometrissä keskeinen komponentti on säteenjakaja. Sellainen voidaan rakentaa esimerkiksi prismoista [3, s. 127] tai piillä päällystetystä kvartsista [1, s. 62]. Interferometrissä valonsäde ohjataan säteenjakajaan, joka jakaa säteen kahteen eri suuntaan. Kun näihin suuntiin sijoitetaan säteeseen nähden kohtisuorat peilit, säteet heijastuvat takaisin säteenjakajalle ja siitä detektorille. Jos peilit ovat eri etäisyyksillä säteenjakajasta, syntyy säteenjakajalle palaavien säteiden välille vaihe-ero, joka aiheuttaa interferenssiä. Näin interferenssin määrää voidaan kontrolloida peilejä liikuttamalla. [4, s. 57.] Michelson-interferometri esitetään kuvassa 6.



Kuva 6. Michelson-interferometri

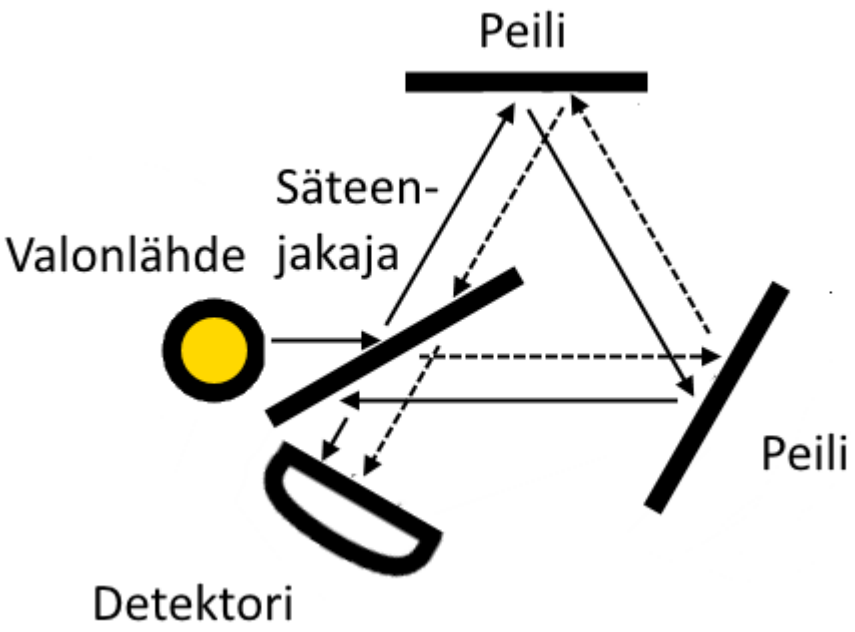
Mach-Zehnder-interferometrissä on kaksi peiliä ja kaksi säteenjakajaa aseteltuna kuten kuvassa 7. Ensimmäisellä säteenjakajalla säde jaetaan kahteen haaraan. Nämä kaksi sädettä ohjataan peileillä toiselle säteenjakajalle, joka yhdistää säteet. Interferenssin voi saada aikaiseksi esimerkiksi kääntämällä toista peiliä hieman [3, s. 411] tai sijoittamalla yhden säteen tielle stabiileissa olosuhteissa pidettävän referenssinäytteen ja toisen säteen tielle näytteen, jonka optisia ominaisuuksia voi muuttaa esimerkiksi paineen tai lämpötilan avulla. Näin toiseen säteeseen saadaan aikaiseksi muutos, joka tuottaa interferenssiä. [4, s. 73.] Koska valonsäteet kulkevat eri reittejä, optiikan kohdis-

taminen voi olla haasteellista. [3, s. 411] Interferometrin voi toteuttaa peilien sijaan myös optisilla kuiduilla [4, s. 73].



Kuva 7. Mach-Zehnder-interferometri

Sagnac-interferometrissä on yksi säteenjakaja ja vähintään kaksi peiliä tai optinen kuitu. Lähteestä tuleva valo jaetaan kahteen eri suuntaan. Valonsäteet kiertävät saman silmukan vastakkaisiin suuntiin palaten lopulta säteenjakajalle ja kulkeutuen siitä detektorille. Koska säteet kulkevat molemmat samaa reittiä, niitä ei voi erottaa toisistaan, eikä interferenssiä saada aikaiseksi perinteisin menetelmin. [3, s. 412.] Mikäli peilien sijaan käytetään kuitua, Sagnac-interferometri havaitsee herkästi kuidussa kulkevan valonsäteen kulkusuuntaa myötäilevän tai vastustavan pyörimisliikkeen, jolloin sitä voi käyttää gyroskooppina. [4, s. 74]. Sagnac-interferometri esitetään kuvassa 8.

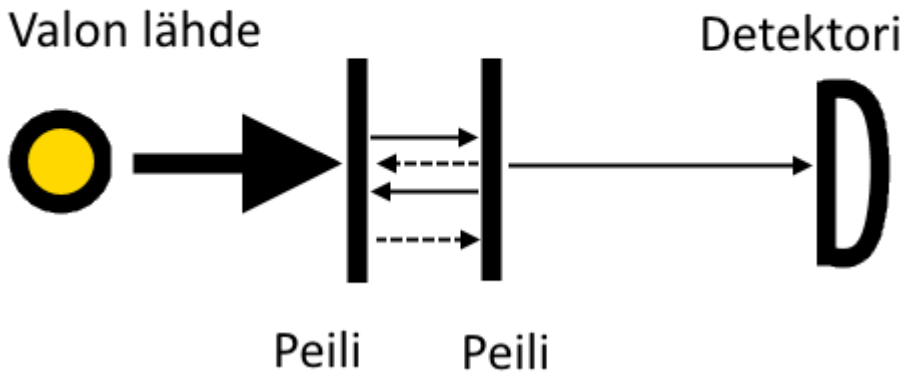


Kuva 8. Sagnac-interferometri

Kaikki edellä kuvatut interferometrityypit vaativat siis säteenjakajan toimiakseen. Fabry-Perot-interferometri poikkeaa tästä koostuen pelkästään peileistä.

2.1 Fabry-Perot-interferometri

Fabry-Perot-interferometrin (FPI) kehittivät Charles Fabry ja Alfred Perot 1800-luvun lopulla [3, s. 421]. Se koostuu kahdesta vastakkain asetetusta, osittain läpinäkyvästä peilipinnasta, jotka heijastavat valonsädettä edestakaisin. Rakenne esitetään kuvassa 9. Osa aallonpituuksista voimistuu interferenssin myötä ja poistuu peilien välistä, kun taas muut aallonpituudet vaimenevat. Perinteisesti pinnat ovat olleet metalloituja, esimerkiksi hopeaa, mutta nykyisin niitä tehdään myös ohutkalvotekniikalla kerrostamalla.

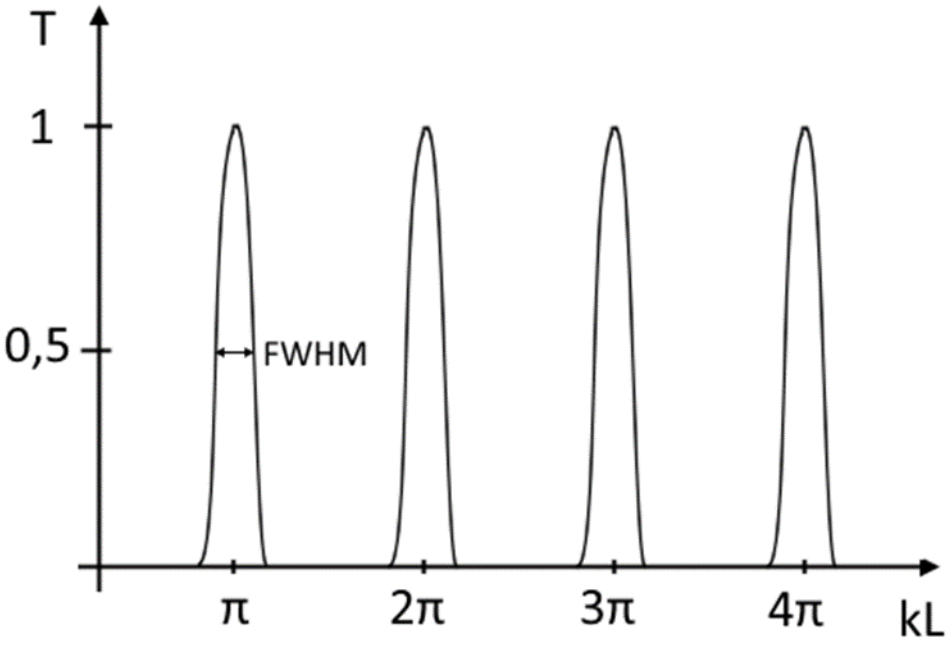


Kuva 9. Fabry-Perot-interferometri

Jos toista peiliä on mahdollista liikuttaa, kyseessä on interferometri. Jos peilit pidetään vakioetäisyydellä toisistaan, elementtiä kutsutaan etaloniksi. Interferometrissä peilipintojen välissä on tyypillisesti ilmaa, mutta etalonin tapauksessa se voi olla jotain muutakin ainetta. [4, s. 421.]

Interferometrin transmissio noudattaa yhtälöä $T = \frac{1}{1+Fsin^2(kL)}$, missä k on aaltoluku väliaineessa, L peilipintojen etäisyys toisistaan ja finessikerroin $F = \left(\frac{2R}{1-R^2}\right)^2$ sillä oletuksella, että molempien peilipintojen reflektanssi on R.

Transmission yhtälöä kutsutaan myös Airy-funktioksi. Sen maksimit ovat kohdissa, joissa $sin^2(kL) = 0$. Tämä toteutuu, kun $kL = m\pi$, missä m on mikä tahansa kokonaisluku. [3, s. 75–76.] Karkea hahmotelma funktiosta esitetään kuvassa 10.



Kuva 10. Airy-funktio

Koska k on aaltoluku väliaineessa ja kyseinen suure on riippuvainen väliaineen taitekertoimesta, interferometrin läpäiseviin aallonpituuksiin voi vaikuttaa joko muuttamalla peilien etäisyyttä tai väliainetta. Näistä vaihtoehdoista ensimmäinen on yleensä helpompi toteuttaa kuin jälkimmäinen.

Transmissiopiikin muoto vaikuttaa spektrometrin resoluutioon. FWHM (Full Width Half Maximum) eli piikin leveys puolessa välissä piikin maksimikorkeutta halutaan yleensä pitää mahdollisimman kapeana, transmissio niin lähellä sataa prosenttia kuin mahdollista ja molemmat ominaisuudet mahdollisimman vakioina koko spektrometrin skannausvälillä. Mitä korkeampi peilien reflektanssi, sitä kapeampi piikki. Mitä kapeampi piikki, sitä korkeampi resoluutio.

Spektroskopiassa hyvästä signaali-kohina-suhteesta on apua signaalien heikkouden takia. Sitä voi parantaa keräämällä enemmän valoa sensorille esim. suurentamalla apertuureja [6, s. 16].

2.2 MEMS

Mikrovalmistusteknologia on lähtöisin elektronikkateollisuuden tarpeesta valmistaa yhä pienempiä integroituja piirejä. MEMS-järjestelmissä mikroskooppisten rakenteiden sähköisten ominaisuuksien lisäksi hyödynnetään myös niiden mekaanisia ominaisuuksia.

MEMS-osien valmistus perustuu materiaalin lisäämiseen ja/tai poistamiseen kerros kerrokselta. Kasvatusalustaa kutsutaan substraatiksi ja se on perinteisesti ollut piitä. Prosessi voi olla joko litografinen tai ei-litografinen. Litografisessa valmistusprosessissa osa muokattavasta materiaalista peitetään suojaavalla kerroksella, jonka jälkeen suojaamattomille alueille joko kasvatetaan uutta materiaalia tai etsataan materiaalia pois. [5, s. 33.] Ei-litografisissa menetelmissä suojaavia kerroksia ei käytetä, vaan materiaalia muokataan suoraan esimerkiksi laser-säteellä [5, s. 63]. Valmistusmenetelmiä on useita, eikä niitä käydä läpi tässä dokumentissa sen tarkemmin.

Bulk-MEMS:issä mikroskooppiset rakenteet valmistetaan kaivautumalla substraatin sisään. Pii sopii tähän tarkoitukseen hyvin johtuen sen sähkömekaanisista ominaisuuksista ja sopivuudesta etsaukseen. Bulk-FPI:ssä peilit voidaan tehdä metalloimalla. Pinta-MEMS:issä taas rakenteet muodostetaan lisäämällä kerroksia substraatin päälle, ja substraatin materiaalivalinta on täten vapaampi kuin bulk-MEMS:issä. [6, s. 17.]

Pinta-MEMS:illä toteutetussa FPI:ssä peilit ovat usein DBR (Distributed Bragg Reflector) -pintoja. Niissä pinnoille on kasvatettu vuoron perään kahdesta eri taitekertoimisesta materiaalista kerroksia, joiden paksuus on neljännes aallonpituudesta. Jokainen materiaalien rajapinta heijastaa osan valosta takaisin. Mitä enemmän näitä pareja kasvatetaan ja mitä suurempi näiden taitekertoimien ero on, sitä heijastavampi peili saadaan aikaiseksi. [7.]

Optiset apertuurit voivat olla muutamasta mikrometristä useampaan millimetriin. Pinta-MEMS:illä toteutetussa FPI:ssä isommat apertuurit vaativat ylimääräisiä tukirakenteita pitämään DBR-peilit tasomaisena.

Yleisin tapa aktuoida FPI:n MEMS-rakenteita on käyttää sähköstaattista voimaa kahden samansuuntaisen levyn liikuttamiseksi lähemmäksi toisiaan. Tyypillisesti tällä tavalla aktuoitu peili kykenee liikkumaan kolmanneksen ilmaraosta ennen kuin peilien

välinen vetovoima vetää peilin kiinni alempaan elektrodiin. Aktuointiin käytetään yleensä 10-50 voltin jännitettä.

Toinen tapa on käyttää pietsoaktuointia. Sen avulla on tehty FPI:tä, joissa peilit ovat eri kiekoilta peräisin. Aktuointimekanismi ei tällöin ole varsinaisesti MEMS-prosessin tuotosta, joten tällaista järjestelmää ei välttämättä lasketa MEMS-FPI:ksi. [6, s. 17.]

2.3 Kemometria

Kemometria on analyyttisen kemian osa-alue. Analyyttisessä kemiassa tutkitaan näytteitä ja pyritään tuottamaan niistä jotain hyödyllistä informaatiota. Kemometriassa tämä tehdään tilastollisten menetelmien kautta. Se syntyi 1970-luvulla tietokoneavusteisen laskennan yleistyttyä ja on kasvattanut merkitystään datamäärien kasvaessa. Erityisesti spektroskopian tuottamien datamäärien käsittelyssä tilastollisista menetelmistä on apua. [8, s. 2.]

Spektroskooppinen mittausdata koostuu tiedosta ja kohinasta. Tietoa on se, mitä näytteestä halutaan saada irti ja kohinaa taas se, mistä kemometrisin menetelmin halutaan päästä eroon. [9, s. 2.] Kemometriassa tieto irroitetaan mittausdatasta matemaattisen mallin avulla. Tämä malli on aina approksimaatio ja sillä on rajallinen pätevyys. Malli tulee aina validoida ennen käyttöä. [9, s. 20.]

Data voidaan esikäsitellä ennen analyysia. Esikäsittelyssä yleensä yritetään korjata datan vinoumat. Sen jälkeen dataa käsitellään regressioanalyysin keinoin. Keinoja ovat mm. PCA (Principal Component Analysis) ja PCR (Principal Component Regression), joissa molemmissa muodostetaan datasta ortogonaalisia komponentteja, kunnes datan varianssi on saatu kokonaan käsiteltyä. Tarkempi menetelmien kuvaus jätetään tämän työn ulkopuolelle. [9, s. 32.]

Työssä hyödynnetään valmista kemometrista mallia näytteen konsentraation laskemiseen spektristä. Kemometrisessa mallissa on määritelty mitatuille aallonpituuksille B-kertoimet, jotka ovat eräänlaisia painokertoimia. Konsentraatio saadaan selville laskemalla pistetulo B-kertoimet sisältävän vektorin ja mitatun spektrin absorbanssi-arvojen välillä.

Absorbanssiarvot voidaan laskea Beerin ja Lambertin lailla, joka voidaan esittää muodossa $A = -log_{10}\frac{P}{P_0}$. P on näytteen kanssa mitattu intensiteetti ja P_0 on intensiteetti ilman näytettä. Molemmista arvoista on vähennetty pimeätaso eli detektorin signaali ilman valoa eli kohina.

Laki on mahdollista esittää myös muodossa $A = b \times c \times \varepsilon$, jossa b on valon kulkumatka näytteessä, c näytteen konsentraatio ja ε materiaalille ominainen molaarinen absorptiokerroin. [9, s. 8.] Konsentraatio on siis mahdollista laskea absorbanssista, jos tiedetään kyvetin pituus sekä mitattavan aineen molaarinen absorptiokerroin. Tilannetta hankaloittaa kuitenkin usein se, että useamman näytteessä olevan aineen absorbanssipiikit osuvat osittain toistensa päälle. Ongelman saa ratkaistua tarkastelemalla riittävän montaa aallonpituutta.

3 Tietotekniikka

Työssä käytetään Raspberry Pi-minitietokonetta, johon on asennettu pieni kosketusnäyttö. Anturi kytkeytyy tietokoneeseen USB (Universal Serial Bus) -väylän kautta ja näkyy käyttöjärjestelmälle sarjaporttina. Ohjelmalle tehdään käyttöliittymä Qt-kirjastolla.

3.1 Raspberry Pi

Raspberry Pi on tietokone, jonka erityispiirteitä ovat edullisuus ja pienikokoisuus. Laite maksaa alle 50 dollaria ja vastaa kooltaan luottokorttia. Sen alkuperäinen tarkoitus on ollut tarjota halpa tietotekniikan ja ohjelmoinnin oppimisalusta erityisesti lapsille, mutta laitetta käytetään ahkerasti myös harrastuskäyttöön. [10.] Ensimmäisen sukupolven Raspberry Pi:t pohjautuvat Broadcomin ARM (Advanced RISC Machines) -pohjaiseen 800 megahertsin BCM2835-prosessoriin, kun taas uudempi Raspberry Pi 2 käyttää saman valmistajan neliytimistä BCM2836-prosessoria. Raspberry Pi sisältää tyypillisimmät nykyaikaisista tietokoneista löytyvät liitännät kuten USB-, Ethernet- ja HDMI (High Definition Multimedia Interface) -liitännät. [11.]

Kuva 11. Raspberry Pi

Tavallisesta tietokoneesta poiketen Raspberry Pi, joka näkyy kuvassa 11, sisältää myös piikkiriman, jonka kautta tietokoneen käyttäjä pääsee käsiksi osaan prosessorin input- ja output-pinneistä. Raspberry Pi siis sijaitsee ominaisuuksiltaan jossain tavallisen tietokoneen ja Arduino-tyyppisten mikro-ohjainlevyjen välimaastossa. Laitteen suunnittelusta ja valmistuksesta vastaa Raspberry Pi Foundation.

3.1.1 Raspbian

Suosituin käyttöjärjestelmä Raspberry Pi:lle on Raspbian, joka on liukulukuoperaatioiden osalta optimoitu Raspberry Pi:n laitteistolle. Raspberry Pi Foundationin aikaisemmin julkaisema käyttöjärjestelmä oli tarkoitettu yksinkertaisemmille ARM-laitteille, jotka eivät kykene laitetason liukulukuoperaatioihin ja joissa ne tehdään sen takia ohjelmistossa. Tämä hidastaa suoritusta huomattavasti. Koska Raspberry Pi kykenee laitetason liukulukulaskentaan, ryhmä harrastajia käänsi laitteelle Debian Linuxin Wheezy armhf -version, joka sallii sen. Raspbian perustuu siis Debianiin, joka on Linux-distribuutio. Raspbiania kehittävä vapaaehtoisryhmä on kääntänyt Raspbianille yli 35 000 ohjelmis-

topakettia, jonka ansioista käyttöjärjestelmälle on laaja kirjo ohjelmia, jotka on helppo asentaa. [12.]

3.1.2 PiTFT

Tyypillisesti Raspberry Pin kanssa käytetään tavallista tietokoneen näyttöä tai televisiota HDMI-liitännän kautta. Tässä projektissa oli kuitenkin tarkoituksena rakentaa pienikokoinen ja mahdollisesti jopa kannettava järjestelmä, johon täysikokoinen näyttö ei sovellu. Näytöksi valittiin Adafruitin 2,8 tuuman PiTFT-näyttö, joka on pienikokoinen Raspberry Pi:n GPIO (General Purpose Input/Output) -piikkirimaan kytkettävä resistiivinen kosketusnäyttö. Tietokone ohjaa näyttöä SPI (Serial Peripheral Interface Bus) -väylän kautta. Näyttömoduulissa on myös paikat neljälle fyysiselle napille. Moduuli esitetään kuvassa 12.

Kuva 12. PiTFT-kosketusnäyttö

Näytön resoluutio on 320x240, mikä hankaloittaa perinteisen graafisen käyttöliittymän käyttöä näytön kautta, koska suurin osa ohjelmista olettaa tarjolla olevan enemmän näyttötilaa. [13.] Työskentely laitteella onkin järkevämpää hoitaa etäyhteyden kautta esimerkiksi TightVNC-etätyöpöytäyhteyttä hyödyntäen.

SPI-väylä on myös verrattain hidas näyttöruudun tarpeisiin. Yhteys on niin hidas, että näyttöön ei ole katsottu tarpeelliseksi rakentaa OpenGL-tukea. HDMI-väylän lisäksi kolmas mahdollisuus näytön liitännäksi on Raspberry Pi:ltä löytyvä 15-pinninen DSI (Display Serial Interface) -liitin. Se kytkeytyy prosessorin MIPI (Mobile Industry Processor Interface) -rajapintaan. Tätä liitäntää käyttäviä näyttöjä on kuitenkin huonommin tarjolla ja PiTFT houkutteli myös sillä, että sen sai kiinteästi liitettyä Raspberry Pi:n päälle, kun taas DSI-liitäntäiset näytöt kytkeytyvät laitteeseen pelkällä lattakaapelilla, jolloin niiden kiinnityksestä Raspberry Pi:hin saa huolehtia itse. [14.]

Pelkkä näytön kytkeminen piikkirimaan ei saa tietokoneen kuvaa ohjautumaan kosketusnäytölle, vaan käyttöjärjestelmään täytyy asentaa kernel-moduuleja, jotka ohjaavat Linuxin framebufferin SPI-väylän kautta näytölle. Adafruit tarjoaa valmiin skriptin, joka lataa Internetistä oikeat moduulit ja asentaa ne. Kun tietokoneen käynnistää seuraavan kerran, kuva ohjautuu kosketusnäytölle. [13.]

Adafruitin ohjeet olettavat, että Raspberry Pi käynnistyy komentotilaan, jossa sitten ajetaan startx-komento graafisen käyttöliittymän käynnistämiseksi. Raspberry Pi on kuitenkin myös mahdollista konfiguroida käynnistymään automaattisesti graafiseen käyttöliittymään, josta seuraa se, että kuva ohjataan takaisin HDMI-liittimelle ja PiTFT-näyttö pimenee graafisen käyttöliittymän käynnistyessä. Ongelman saa korjattua seuraavasti:

Ensin ajetaan komentorivillä komento "sudo apt-get install xserver-xorg-video-fbdev". Tämä asentaa Linux framebuffer devicen. Sitten luodaan sille konfigurointitiedosto "/usr/share/X11/xorg.conf.d/99-fbdev.conf" ja lisätään sinne seuraava sisältö:

```
Section "Device"
          Identifier "myfb"
          Driver "fbdev"
          Option "fbdev" "/dev/fb1"
EndSection
```

Näiden muutosten jälkeen graafinen käyttöliittymä ohjautuu PiTFT-näytölle myös siinä tapauksessa, että Raspberry Pi on konfiguroitu käynnistymään suoraan graafiseen tilaan. [15.]

3.2 Sarjaportti

Sarjakommunikaatiossa data lähetetään ja vastaanotetaan yksi bitti kerrallaan. Yhteyden nopeus määritellään bitteinä sekunnissa (bps) tai perinteikkäämmin baudeina, jossa lasketaan sähköisen signaalin muutosnopeutta sekuntia kohden. Jos signaalilla on vain kaksi mahdollista tilaa, sama määrä baudeja ja bittejä sekunnissa kuvaa samaa nopeutta. Muussa tapauksessa kyse on eri nopeuksista.

Tietokoneen sarjaportti noudattaa RS-232-protokollaa. Se määrittelee 25-pinnisen liitännän, josta suurin osa on erilaisille hallintasignaaleille varattuja. RS-232 on full-duplex, mikä tarkoittaa sitä, että linjan yli on mahdollista lähettää ja vastaanottaa tietoa samanaikaisesti.

Sarjaportti voi lähettää ja ottaa vastaan dataa synkronisesti tai asynkronisesti. Asynkronisessa kommunikaatiossa datalinja pidetään ykkösessä, kunnes merkki halutaan lähettää. Tällöin linja asetetaan nollaan. Tämä on aloitusbitti. Välittömästi sen jälkeen lähetetään varsinaiset databitit ja niiden jälkeen mahdollinen pariteettibitti. Lopuksi lähetetään yksi tai useampi lopetusbitti. Ne ovat aina ykkösiä.

Pariteettibitti on databittien summa. Koska se on vain yksi bitti, se kykenee kertomaan sen, onko summa parillinen vai pariton. Jos ykkösiä on parillinen määrä, pariteettibitti on nolla. Jos ykkösiä on pariton määrä, pariteettibitti on yksi. Summa lasketaan sekä lähetettäessä että vastaanottaessa ja tulosten tulee täsmätä. Jos ne eivät täsmää, lähetyksessä on tapahtunut virhe.

Synkronisessa kommunikaatiossa bittejä kulkee linjalla jatkuvasti kellopulssin tahdissa. Kellopulssi pitää tällöin välittää sekä lähettäjälle että vastaanottajalle. Datalähetyksessä käytetään yleensä jonkinlaista pakettiprotokollaa. Protokolla määrittelee datan alun ja lopun, sekä bittisekvenssin, jota lähetetään silloin, kun linjalla ei kulje varsinaista dataa. [16.]

3.2.1 Sarjaportti tiedostona

Linuxissa laitteiden kanssa kommunikoidaan kuin ne olisivat tiedostoja. Tiedoston nimi voi vaihdella kytkettävän laitteen mukaan. Sarjaporttina näkyvät USB-laitteet ovat usein joko "/dev/ttyUSBx" tai "/dev/ttyACMx" -nimisiä tiedostoja, joissa x on nollasta lähtevä juokseva numerointi. Nimitys riippuu siitä, minkä rajapinnan USB-ohjain toteuttaa. [17.]

Koska laitteet näkyvät Linuxissa tiedostoina, niiden kanssa on periaatteessa mahdollista kommunikoida avaamalla kaksi terminaalia ja ajamalla toisessa echo-ohjelmaa ja toisessa cat-ohjelmaa. Echo syöttää tiedostoon tekstiä ja cat lukee tekstiä tiedostosta. Sarjakommunikaatio ei siis periaatteessa vaadi erillistä kommunikaatio-ohjelmaa. Ilman tarkoituksenmukaista ohjelmaa sarjakommunikaatio ei kuitenkaan useimmin toimi, koska sarjaportin oletusasetukset eivät ole asianmukaiset. Asetukset on mahdollista asettaa stty-ohjelmalla.

Kommunikointiin laitteen kanssa ohjelmallisesti riittää periaatteessa tavallinen tiedosto-I/O. C++-kielessä tämän voi toteuttaa mm. filestream-objektin kautta. Tämäkin kuitenkin vaatii sarjaportin konfiguroimista toimiakseen.

Tyypillisesti Linuxissa sarjaporttikommunikaatio toteutetaan matalan tason C-komentoja käyttäen. Tiedosto avataan open()-funktiolla. Funktiolle annetaan parametrina tiedoston nimi sekä asetetaan erilaisia lippuja koskien tiedoston käsittelyä. Yleisimmin käytetyt liput koskevat sitä, avataanko tiedosto kirjoittamista (O_WRONLY), lukemista (O_RDONLY) vai molempia (O_RDWR) varten. O_NOCTTY-lippu kertoo Linuxille, että ohjelma ei halua olla hallitseva terminaali kyseiselle portille. O_NDELAY-lippu kertoo, että ohjelma ei välitä DCD (Data Carrier Detect) -signaalin tilasta. Jos tätä lippua ei aseteta, ohjelma odottaa, kunnes DCD-linja on kunnossa. DCD-signaali kertoo, onko datalinja käytössä. Aina tätä signaalia ei kuitenkaan käytetä, joten sen odottaminen saattaa pysäyttää ohjelman lopullisesti.

Kirjoittaminen tapahtuu write()-komennolla. Funktio palauttaa lähetettyjen bittien määrän. Jos funktio palauttaa negatiivisen arvon, lähetyksessä on tapahtunut jokin virhe.

Lukeminen read()-funktiolla palauttaa ne bitit, joita portissa on tarjolla. Jos bittejä ei tule, funktio odottaa niitä ikuisesti, jos lukemiselle ei ole määritetty mitään aikarajaa.

fcntl(fd, F_SETFL, FNDELAY)-käsky muuttaa read()-funktion käyttäytymistä. FNDELAY-lippu saa read()-funktion palauttamaan välittömästi nollan, jos dataa ei ole luettavissa. Normaalin toiminnan saa palautettua kutsumalla fcntl(fd, F_SETFL, 0). ”fd” on kokonaisluku, joka kertoo, mitä tiedostoa ollaan käsittelemässä.

3.2.2 Termios

Sarjaportti konfiguroidaan käyttämällä POSIX (Portable Operating System Interface) -terminaalirajapintaa. Sen saa käyttöön omassa ohjelmassa liittämällä termios.h-tiedoston mukaan.

tcgetattr()-funktio hakee sarjaportilta sen tämänhetkiset asetukset ja tcsetattr()-funktio asettaa sarjaportille uudet asetukset. Molemmat käyttävät toimintaansa termios-struktuuria, joka sisältää portin asetukset. Struktuuri koostuu bittitaulukoista, joiden bittejä asetetaan valmiiksi määriteltyjen vakioiden avulla. Taulukossa 1 listataan struktruurin jäsenet.

Taulukko 1. Termios-struktuurin jäsenet

Struktuurin jäsenet	**Kuvaus**
c_cflag	Yleiset asetukset
c_lflag	Linja-asetukset
c_iflag	Sisääntuloasetukset
c_oflag	Ulostuloasetukset
c_cc	Ohjausmerkit
c_ispeed	Sisääntulonopeus

c_cflag asettaa baudinopeuden, databittien määrän, paritettibitit, lopetusbitit ja laitteistopohjaisen flow controllin. Flow controllilla hallitaan sitä, milloin dataa saa linjalla lähettää. Keskeisiä lippuja ovat CREAD ja CS8. CREAD kertoo, että sarjaporttiajuri lukee sisääntulevat bitit ja CS8 asettaa merkin kooksi kahdeksan bittiä.

c_lflag määrää sen, miten sisääntuleviin merkkeihin suhtaudutaan. Tällä jäsenellä päätetään, käytetäänkö canonical- vai non-canonical-tilaa. Canonical-moodissa lukeminen

tapahtuu riveittäin. Dataa ei siis lueta ennen kuin puskuriin ilmestyy rivinvaihtomerkki. Non-canonical-moodissa luetaan raakadataa. Luettavien merkkien määrä on tällöin ennalta määrätty. Non-canonical-moodissa on myös mahdollista asettaa ajastin, jonka lauetessa luetaan vastaanotetut merkit, vaikka niitä ei olisikaan vielä haluttua määrää.

Tyypillisesti canonical-tilassa käytetään lippuja ICANON, ECHO ja ECHOE. ICANON asettaa canonical-tilan päälle. ECHO ja ECHOE toistavat sisääntulevat merkit. Non-canonical-tilaa varten asetetaan samat liput, mutta negaationa.

c_iflag määrittelee mahdollisen sisääntulevalle datalle tehtävän käsittelyn. Tähän kuuluu mm. pariteettivirheiden ja syötteenohjausmerkkien käsittely, sekä ohjelmistopohjainen flow control.

c_oflag kontrolloi ulostuloa. Suurin osa siitä koskee erilaisten ylimääräisten taukojen pitämistä. Se on perua ajoilta, jolloin sarjaporttiin kytketyt laitteet saattoivat olla niin hitaita, että niillä oli vaikeuksia pysyä kommunikaatiossa mukana.

c_cc on char-taulukko, joka sisältää kontrollimerkkien määrittelyt, vähimmän luettavan merkkimäärän sekä ajastimen. VTIME määrittelee ajastimen maksimiajan, jonka verran uutta dataa odotetaan. Aika ilmoitetaan sekunnin kymmenyksissä. VMIN sisältää pienimmän luettavan merkkimäärän.

Sekä tcgetattr() että tcsetattr() molemmat käyttävät alemman tason systeemikutsua ioctl(). Sitä käytetään I/O-laitteiden (Input/Output) hallintaan. Sen prototyyppi on (int fd, int request, ...). Ensimmäinen argumentti viittaa haluttuun laitteeseen ja toinen argumentti kertoo, mitä laitteelle halutaan tehdä. Kolmas argumentti voi olla joko kokonaisluku, joka lähetetään laitteen ajurille, tai osoitin, jolla lähetetään tai vastaanotetaan dataa ajurilta. Sen tyyppi ja sisältö riippuvat toimenpiteestä, jonka käskyn halutaan suorittavan.

termios.h määrittelee sarjaportin käyttöön liittyviä vakioita request-parametrille. Erityisen käyttökelpoinen, ja tässäkin työssä käytetty, on FIONREAD. Sille ei ole omaa POSIX-funktiota, minkä takia ioctl()-funktio ja sen käyttö on tarpeellista käsitellä. ioctl(fd, FIONREAD, &bytes_at_port)-käsky sijoittaa bytes_at_port-muuttujaan sarjaportissa luettavissa olevien bittien määrän ilman, että bittejä tarvitsee vielä lukea. Non-

canonical-moodissa tämä tarkoittaa sitä, että lukemista ei tarvitse tehdä ennen kuin VMIN-muuttujan osoittama minimimerkkimäärä on jo puskurissa.

Vanhempi materiaali suosittelee myös select()-systeemikutsun käyttämistä. Sen ovat käytännössä korvanneet uudemmat poll() ja epoll(). Nämä kutsut ovat hyödyllisempiä verkkoviestinnässä, jossa samanaikaisia yhteyksiä voi olla tuhansia. Näiden kutsujen tärkein etu on siinä, että niillä kykenee kurkistamaan, onko terminaaliin tullut syötettä ilman, että sitä syötettä tarvitsisi vielä siinä vaiheessa ruveta lukemaan. [18.]

3.3 Qt

Qt on alustariippumattomien graafisten käyttöliittymien tuottamiseen tarkoitettu työkalupakki. Trolltech julkisti Qt:n ensimmäisen version vuonna 1995. [19, s. 13].

Nokia osti Trolltechin ja samalla Qt:n kehityksen vuonna 2008. Nokia myi Qt:n Digialle paloittain vuosina 2011-2012. Digia siirsi Qt:n työstämisen vuonna 2014 tytäryhtiölleen Qt Companylle. Qt Company kehittää Qt:tä yhdessä Qt Projectin kanssa. Qt Project koostuu yksittäisistä kehittäjistä ja yrityksistä. [20.]

Qt vaatii ohjelmakoodiin omia C++-standardin ulkopuolisia lisäyksiään. C++-kääntäjä ei ymmärrä niistä mitään, vaan ohjelmakooditiedosto on ensin annettava metaobjektikääntäjälle käsiteltäväksi. Qt:n mukana tulee ohjelma nimeltä qmake, joka tekee tämän ohjelmoijan puolesta.

Käyttöliittymän voi tehdä Qt:ssä kahdella eri tavalla. Jokaisen elementin, layoutin, signaalin jne. voi joko määritellä ohjelmakoodissa tai käyttää Qt Designeria, jolla lomakepohjaisen käyttöliittymän kykenee suunnittelemaan graafisesti vetämällä hiirellä elementtejä lomakkeelle.

Jos käyttöliittymän suunnittelee Qt Designerilla, ohjelma luo .ui-päätteisen tiedoston, jossa käyttöliittymän kaikki elementit ovat listattuna XML-muodossa. Kyseinen tiedosto muunnetaan C++-koodiksi joko kääntämisen tai ajon aikana. Jälkimmäisestä tavasta voi olla hyötyä, jos käyttöliittymä halutaan generoida dynaamisesti.

Työtä tehdessä viimeisin Qt:n versio oli Qt 5.1. Alun perin oli tarkoitus käyttää tätä uusinta versiota. ARM-pohjaiselle Raspberry Pi:lle ei ollut vielä valmiita Qt 5 -paketteja, joten Qt 5 piti kääntää itse. Koko työkalupakin kääntäminen on melko raskas prosessi ja se vie useamman vuorokauden, jos sen tekee Raspberry Pi:llä. Julkisesti on tarjolla työkalut ns. ristikääntämiseen (cross-compiling), jossa itse kääntäminen tehdään jollain tehokkaammalla tietokoneella, mutta käännöksen tulos toimii Raspberry Pi:llä. Tehokkaalla, pelikäyttöön tarkoitetulla kannettavalla tietokoneella kääntäminen kesti silti useita tunteja, tosin kääntäminen tapahtui langattomasti lähiverkon yli.

Kääntäminen periaatteessa onnistui, mutta myöhemmin kävi ilmi, että koska PiTFT-näyttömoduuli toimii verrattain hitaan SPI-väylän kautta, näyttömoduulin tekijät eivät kokeneet järkeväksi implementoida sille OpenGL:ää. Qt 5 -kirjaston yhden uuden piirteen ollessa nimenomaisesti se, että widgetit ovat OpenGL-pohjaisia, oli helpointa vain käyttää vanhempaa Qt 4.8 -versiota. Tässä valinnassa oli myös se etu, että Qt 4.8 -paketit löytyivät valmiiksi Raspberry Pi:n pakettikirjastosta, joten ne oli helppo ottaa käyttöön.

3.3.1 Metaobjektikääntäjä

Qt laajentaa tavallista C++-kieltä makroilla. Tärkein esimerkki tästä ovat metaobjektit. Ne mahdollistavat Qt:n signaali- ja slottijärjestelmän. Qt:n metaobjekti-järjestelmä koostuu kolmesta elementistä: QObject-luokasta, Q_OBJECT-makrosta ja metaobjektikääntäjästä (MOC).

QObject-luokka on Qt:n objektijärjestelmän perusta. Se mahdollistaa signaalien ja slottien käytön luokille, jotka perivät sen. Se myös liittää perivän luokan Qt:n event-järjestelmään. QObjectin avulla objekteista syntyy hierarkioita, joissa objekti tuhoaa automaattisesti destruktorissaan siihen kytketyt lapsiluokat. Kaikki Qt:n widgetit perivät QObjectin.

QOBJECT-makro tulee lisätä sellaisen luokan private-osioon, joka käyttää signaaleja ja slotteja tai jotain muuta Qt:n metaobjektijärjestelmän osaa. Luokan täytyy periä QObject-luokka. [21.]

Metaobjekti-kääntäjä käy läpi C++-koodia sisältävät tiedostot ja löytäessään luokkamäärittelyn, jossa on sisällä Q_OBJECT-makro, se generoi uuden C++-

lähdekooditiedoston, joka sisältää luokan metaobjektikoodin. Tämä uusi lähdekooditiedosto tulee kääntää ja linkata alkuperäisen luokan kanssa. Kaikkein helpointa on antaa qmake-ohjelman luoda makefilet, jolloin se huolehtii MOC:in kutsumisesta.

MOC toteuttaa signaalien ja slottien lisäksi myös alustariippumattoman version tietyistä kääntäjistä löytyvästä property-järjestelmästä. Q_PROPERTY()-makro määrittelee propertyn ja Q_ENUMS()-makro listan enum-tyyppejä, joita property-järjestelmässä voi käyttää.

MOC:in haittapuolena on se, että se ei ole täysin yhteensopiva kaiken C++-koodin kanssa. Keskeisimpänä rajoituksena on se, että template-tyyppiset luokat eivät saa sisältää signaaleja tai slotteja. Template-luokat ovat luokkia, joilla on geneerinen tyyppi. Yleensä niiden tarkoituksena on operoida datalla sen tyypistä riippumatta.

MOC:in käytössä on myös pienempiä rajoituksia. Moniperinnässä MOC olettaa ensimmäisen ja vain ensimmäisen perityn luokan perivän QObject-luokan. Osoittimet funktioon eivät myöskään saa olla signaali- tai slottiparametreja. Qt:n kehittäjät suosittelevat käyttämään tässä tapauksessa mieluummin perintää. Qobject::connect()-funktio tarkastelee signaali- ja slottiparametreja kirjaimellisesti, joten näiden parametrien nimiavaruuden tulee käydä ilmi connect()-kutsussa. Sisäkkäisissä luokissa ei saa olla signaaleja tai slotteja. Signaalien tai slottien palautusarvot eivät voi olla viittauksia. Luokan signaaliosiossa saa olla vain signaaleja ja slottiosiossa vain slotteja. [22.]

3.3.2 Signaalit ja slotit

Signaalit ja slotit ovat Qt:n keskeisiä ominaisuuksia ja ne erottavat Qt:n monesta muusta vastaavasta graafisesta työkalupakista. Niiden avulla objektit kommunikoivat keskenään.

Graafisissa käyttöliittymissä on yleensä tarve välittää joustavasti tietoa objektien välillä. Muut järjestelmät aikaansaavat tämän callback-funktioilla. Kutsuvalle funktiolle annetaan callback-funktion osoite ja se kutsuu tätä tarpeen vaatiessa.

Qt:n signaalit ja slotit ovat vaihtoehtoinen tapa hoitaa asia. Signaali lähetetään, kun tietyt kriteerit täyttyvät. Widgeteille on määritelty valmiiksi useita signaaleja, joita voi

täydentää itse määrittelemillään signaaleilla. Valmiiksi määriteltyjä signaaleja ovat esimerkiksi napin painamiset tai ikkunan sulkeutuminen.

Slotti on kuin callback-funktio, mutta toisin kuin callback-funktiot, signaali-slotti-mekanismi on tyyppivarma. Se tarkoittaa sitä, että signaalin ja vastaavan slotin argumenttien tulee vastata toisiaan. Toisaalta signaali-slotti-järjestelmä on jonkin verran hitaampi kuin funktio-osoittimilla toteutetut callback-funktiot.

Signaalin lähettävä objekti ei välitä siitä, ottaako mikään objekti signaalia vastaan. Slotti ei myöskään tiedä, onko siihen tulossa yhtään signaalia. Signaalit ja slotit eivät siis ole riippuvaisia toistensa olemassaolosta, mikä on eduksi, kun halutaan kirjoittaa modulaarista koodia.

Yksi signaali voi olla liitoksissa moneen slottiin ja yhteen slottiin voi tulla monta signaalia. Signaaleja voi myös liittää toisiin signaaleihin, jolloin ensimmäisen signaalin lähettäminen saa myös toisen signaalin lähtemään välittömästi.

Signaalit ja slotit kytketään toisiinsa QObject::connect()-funktiolla, jonka prototyyppi on

```
bool QObject::connect(const QObject * sender, const char *
signal, const QObject * receiver, const char * method,
Qt::ConnectionType type = Qt::AutoConnection).
```

Prototyypistä näkee, että lähettäjän ja vastaanottajan tulee kummankin joko olla itse tyyppiä QObject tai periä se. “type” määrittelee signaalin tyypin. Tyyppi määrää sen, suoritetaanko slotti välittömästi signaalin saapuessa vai vasta sitten, kun ohjelman komento on palannut takaisin ohjelman tapahtumat käsittelevään silmukkaan.

Signaaleja ja slotteja ei ole pakko kytkeä toisiinsa ohjelmallisesti, vaan asian voi hoitaa myös Qt Creatorin käyttöliittymäeditorissa.

Signaalit ovat julkisia funktioita, joten niitä on mahdollista lähettää mistä tahansa. Suositus on kuitenkin, että signaali lähetetään vain siitä luokasta, joka signaalin määrittelee, tai sen perivistä luokista.

Kun signaali kutsuu slottia, slotti suoritetaan yleensä välittömästi. Signaalit ja slotit on mahdollista määrittää myös jonottaviksi (queued connections), jolloin slotti suoritetaan

vasta myöhemmin. Jos signaali on liitetty useampaan slottiin, slotit suoritetaan yksi kerrallaan siinä järjestyksessä, jossa ne on liitetty signaaliin.

Slotit ovat tavallisia C++-funktioita ja niitä voi myös käyttää sellaisina. Ne noudattavat pääosin C++-kielen sääntöjä. Signaalien avulla niitä voi kuitenkin kutsua mistä tahansa. [23.]

3.3.3 QcustomPlot

QCustomPlot on kolmannen osapuolen kehittämä Qt-moduuli. Se helpottaa huomattavasti kuvaajien piirtämistä Qt-ympäristössä. Se kykenee piirtämään mm. käyriä, palkkikaavioita ja kaksiulotteisia värikarttoja. Piirakkakaavioiden piirtämiseen se ei kykene ja kehittäjä on selittänyt tämän johtuvan osittain teknisistä ja osittain ideologisista syistä. [24.]

QCustomPlot on GPL (GNU General Public License) -lisenssin alainen, mikä hankaloittaa kaupallista käyttöä, koska lisenssi vaatii lähdekoodin julkistamista. Moduulin kehittäjä pyytää ottamaan yhteyttä, jos moduulia halutaan käyttää kaupallisesti.

Moduulin saa käyttöönsä sisällyttämällä qcustomplot.cpp ja qcustomplot.h -tiedostot Qt-projektin osaksi. Tiedostot ovat melko kookkaita, joten ensimmäinen ohjelman kääntämiskerta on selvästi normaalia pidempi. Rapberry Pi:llä käännettäessä kääntämisen kesto on noin viisitoista minuuttia. Kun QCustomPlotin tiedostot on kerran käännetty, sitä ei yleensä tarvitse enää seuraavilla kääntämiskerroilla tehdä ja oman ohjelman kääntäminen nopeutuu. QCustomPlot on mahdollista lisätä projektiin myös valmiiksi käännettynä shared libraryna, jolloin kääntämistä ei tarvitse tehdä itse.

Qt 5.0:sta eteenpäin projektitiedostoon pitää vielä lisätä QT-muuttujaan "printsupport". Vanhemmissa Qt:n versioissa, kuten tässä projektissa käytetyssä Qt 4.8:ssa, tätä lisäystä ei tarvita.

Kun QCustomPlot on lisätty osaksi projektia, sen voi lisätä ohjelman käyttöliittymään. Tämä tapahtuu lisäämällä Qt Creatorin lomake-editorilla QWidget-tyyppinen objekti halutulle lomakkeelle. Tämän jälkeen klikataan oikealla hiiren napilla tätä lisättyä QWidget-objektia ja valitaan "Promote to...". Tämä avaa uuden ikkunan, johon syötetään uuden luokan nimeksi ("Promoted class name") "QCustomPlot". Otsikkotiedoston

nimi "qcustomplot.h" täydentyy automaattisesti omaan kenttäänsä. Tämän jälkeen painetaan Add-nappia ja sitten Promote-nappia. Tehdyt toimenpiteet eivät muuta QWidgetin ulkonäköä miksikään editorinäkymässä, mutta kun ohjelma ajetaan, QWidgetin paikalle piirtyy tyhjä oletuskuvaaja. [25.]

QCustomPlot-kuvaajan käyttö on yksinkertaista. Seuraavassa oletetaan, että "customPlot" on osoitin lomakkeelle lisättyyn QCustomPlot-objektiin. Ensin kuvaajaan lisätään käyrä "customPlot->addGraph()"-käskyllä. Sitten lisätylle kuvaajalle syötetään x- ja y-arvot käskyllä "customPlot->graph(0)->setData(x, y)", jossa x ja y ovat QVector-tyyppisiä muuttujia ja sisältävät kuvaajan pisteiden x- ja y-arvot.

Tämän jälkeen kuvaajan ulkonäköä on mahdollista muuttaa erilaisilla käskyillä. Esimerkiksi "customPlot->xAxis->setLabel("Aika")" asettaa kuvaajassa vaaka-akselin selitetekstiksi "Aika". "customPlot->yAxis->setRange(-1,1)" taas rajaa kuvaajan pystyakselin näyttämään käyrän välillä [-1, 1].

Kun kuvaajan ulkonäkö on saatu määritettyä halutunlaiseksi, kuvaaja piirretään komennolla "customPlot->replot()". Jos kuvaajan on tarkoitus näyttää ohjelman suorituksen aikana reaaliajassa muuttuvaa dataa, kuvaajaa päivitetään samaa replot-käskyä käyttäen. [26.]

4 Järjestelmän toteutus

Järjestelmä koostuu laitteistosta ja ohjelmistosta. Laitteisto on toistaiseksi hyvin yksinkertainen. Raspberry Pi:hin on liitetty näyttö riviliittimellä ja spektrometrianturi USB-johdolla. Järjestelmä saa virran Raspberry Pi:n muuntajasta.

Yhteen koneen USB-porteista on liitetty Wifi-dongle etäkäytön helpottamiseksi. Jos anturin kytkee Raspberry Pi:hin kiinni ajon aikana, koneen Wifi-yhteys katkeaa. Jos anturin kytkee kiinni ja käynnistää koneen vasta sitten, Wifi-yhteys toimii. Oletettavasti anturin kytkeminen siis aiheuttaa kytkentäilmiön, joka on riittävän merkittävä sekoittamaan koneeseen kytkettyjä USB-laitteita. Tämä voi olla ikävä ongelma, jos anturia on tarkoitus vaihtaa ajon aikana.

Laitteistoon verrattuna järjestelmän ohjelmisto on hieman monimutkaisempi ja vaatii pidemmän selityksen.

4.1 Ohjelmiston yleiskuvaus

Ohjelmistossa on kolme luokkaa: MainWindow, MeasurementDevice ja SerialDevice. MainWindow sisältää suurimman osan Qt-koodia. MeasurementDevice määrittelee abstraktin rajapinnan anturin ohjaamiselle ja SerialDevice sisältää sarjakommunikaatioon vaadittavan koodin. SerialDevice perii MeasurementDevice-luokan. Projektissa on myös main.cpp-tiedosto, joka sisältää ohjelman main-funktion.

Kun C++-kielellä tehty ohjelma ajetaan, suoritus alkaa aina main-funktiosta. Sama sääntö pätee myös Qt:lla tehtyihin ohjelmiin. Qt-ohjelmissa main-funktio on tyypillisesti melko lyhyt, koska Qt-kirjasto hallinnoi suurinta osaa ohjelman toiminnoista.

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
        QApplication a(argc, argv);
        MainWindow w;
        w.setWindowState(w.windowState() ^
        Qt::WindowFullScreen);
        w.show();
        return a.exec();
}
```

Esimerkkikoodi 1. main-funktio

Työssä käytetty main-funktio on esitetty esimerkkikoodi 1:ssä sellaisenaan. Funktio luo MainWindow-objektin ja QApplication-objektin. Pääikkuna konfiguroidaan täyttämään koko näyttöruutu ja asetetaan näkyväksi. Käsky `a.exec()` luovuttaa ohjelman hallinnan Qt:lle. Käskyn palautusarvo välitetään sellaisenaan ohjelman palautusarvoksi.

MainWindow-objekti on QMainWindow-luokan perivä luokka, joka sisältää sovelluskeskeisen koodin. QMainWindow-luokkaa käytetään ohjelman pääikkunan luomiseen. [27.] QApplication-objekti mukauttaa ohjelman käyttäjän käyttöjärjestelmän asetuksiin mm. värien ja fonttien osalta, parsii mahdolliset ohjelmalle annetut komentoriviparametrit sekä käsittelee käyttöliittymän tapahtumat, kuten hiiren klikkaukset ja ikkunan koon muutokset. Ohjelmassa on aina yksi ja vain yksi Qapplication-objekti. [28.]

4.2 Sarjaporttiviestintä

Viestintä anturin kanssa on hajotettu kahteen luokkaan, joista MeasurementDevice määrittelee toiminnallisuuden, joka ei ole kommunikaatioalustasta riippuvaista. Se määrittelee abstraktin rajapinnan anturin kanssa kommunikoimiseen. Se sisältää kaiken anturiin liittyvän tiedon, joka ei ole sidoksissa sarjaporttikommunikaatioon. Ajatus on ollut, että anturin kanssa voi kommunikoida riippumatta siitä, millä teknologialla kommunikaatio tapahtuu.

SerialDevice-luokka huolehtii sarjaportin konfiguroinnista ja käyttämisestä.

4.2.1 MeasurementDevice

MeasurementDevice-luokka sisältää DeviceInformation-struktuurin, joka täytetään anturilta kerätyillä tiedoilla, sekä MeasurementSettings-struktuurin, jossa säilytetään mittauksessa käytettäviä asetuksia. Luokka sisältää myös float-taulukon, johon tallennetaan kurantti mittausdata.

Luokka perii QObject-luokan, jotta signaali- ja slottijärjestelmän saisi luokan käyttöön. Tämä on sikäli harmillista, että tähän periaatteessa Qt:stä erilliseen luokkaan sisältyy vain Qt:lle merkityksellistä koodia. Jos graafinen työkalupakki vaihtuu joksikin muuksi, kyseisen koodin joutuu siivoamaan luokasta pois.

Osa luokan funktioista on virtuaalisia. MeasurementDevice-luokan perivät luokat joutuvat toteuttamaan nämä virtuaaliset funktiot. Luokka sisältää myös BuildDeviceInfo()-funktion, joka parsii sille parametrinä annetun stringin ja täyttää DeviceInformation-struktuurin.

Valtaosa MeasurementDevice-luokan funktioista on erilaisia gettereitä ja settereitä. Ne ovat tyypillisesti yksirivisiä funktioita, jotka joko asettavat arvon tai palauttavat sen. Niitä käytetään, jotta luokan käyttäjä ei pääsisi suoraan käsiksi luokan muuttujiin ja olio-ohjelmoinnille keskeinen kapsulointi säilyisi. Settereiden käytössä on myös se etu, että muuttujaan syötettävän arvon järkevyys voidaan arvioida ennen sijoittamista.

MeasurementDevice::BuildDeviceInfo()-funktio parsii parametrina annetusta string-muuttujasta halutut tiedot. Parsimisessa käytetään std::string-luokan funktioita. std::string::find()-funktiolla etsitään hakusanan paikka tekstissä ja istringstream-luokkaa hyväksikäyttäen irrotetaan sitä vastaava arvo tekstin seasta.

4.2.2 SerialDevice

SerialDevice-luokka perii MeasurementDevice-luokan ja toteuttaa sen virtuaaliset funktiot Initialize(), Configure() ja Measure(). Luokan kentät koostuvat yksinomaan sarjaporttikommunikaatiossa tarvittavista muuttujista ja taulukoista.

Luokka sisältää kaksi erilaista konstruktoria. Toisessa sarjaportin osoite annetaan parametrina, kun taas toinen on parametriton. Parametritonta konstruktoria käytettäessä tulee käyttää luokan AssignDevice()-funktiota ja antaa objektille sitä kautta sarjaportin osoite. Jälkimmäisessä tapauksessa pitää vielä kutsua Initialize()-funktiota.

Tieto lähetetään ja vastaanotetaan omissa funktioissaan. Nämä funktiot ovat luokkamäärittelyn private-osiossa, joten vain luokan omat funktiot pääsevät niihin käsiksi. SerialDevice-objektin kautta ei siis voi lähettää anturille mitä tahansa käskyjä, vaan anturia käskytetään ainoastaan luokan määrittelemällä tavalla.

SerialDevice::Initialize()-funktio alustaa sarjaporttikommunikaation. Funktio yrittää avata sarjaportin tiedoston avaamiseen tarkoitetun open()-funktion avulla. open()-funktio palauttaa kokonaislukuarvon, jolla avattuun tiedostoon viitataan. Jos lukuarvo on negatiivinen, tiedoston avaus on jostain syystä epäonnistunut.

Avaamisen jälkeen tiedostoa käsitellään terminaalina. tcgetattr()-käskyllä tallennetaan termios-struktuuriin terminaalin vanhat asetukset. Uusien asetusten termios-struktuuri alustetaan nollaksi ja bittioperaatioilla asetetaan struktuurin kenttien halutut liput pys-

tyyn. Uudet terminaalin asetukset siirretään terminaalille tcsetattr()-funktiolla. Sarjaportti on nyt valmis keskustelemaan ohjelman kanssa.

SerialDevice::SendData()-funktio kirjoittaa dataa sarjaporttitiedostoon eli lähettää dataa anturille. write()-funktio ymmärtää vain C-tyylisiä stringejä, joten funktiolle annetaan parametrina std::string-luokan data()-funktion palauttama char-taulukko. Tarkalleen ottaen kyseinen char-taulukko ei ole C-tyylinen string, koska se ei pääty merkkiin "\0", mutta tällaisessa tilanteessa kyseinen yksityiskohta ei aiheuta ongelmia.

Datan kirjoittamisen jälkeen kutsutaan funktiota tcdrain(), joka odottaa, kunnes data on saatu lähetettyä. Funktiota käyttämällä voidaan varmistaa, että käsky on lähtenyt ennen kuin aletaan pyytää anturilta siihen vastausta. Funktio tallentaa komennon luokan muuttujaan ”lastcommand”, jotta lukufunktio voi saada selville, millaista dataa anturilta on odotettavissa.

SerialDevice::ReceiveData()-funktio lukee anturilta tulevan datan. ioclt()-funktio kertoo, montako bittiä dataa sarjaportissa on luettavissa. Lukusilmukassa yritetään lukea dataa niin paljon kuin sitä on tarjolla. Silmukka odottaa dataa 200 ms, jonka jälkeen lukeminen lopetetaan, jos ioctl() palauttaa nollan. Lukemisen olisi myös voinut toteuttaa VTIME-ajastinta käyttäen, mutta sen resoluutio on vain 0,1 sekuntia, jolloin sillä ei pääse hallinnoimaan ajankäyttöä mikro- tai edes millisekunttitasolla. Jokaisella silmukan kierroksella lähetetään UpdateGUI-signaali, jotta silmukka ei kokonaan estäisi käyttöliittymän päivittymistä.

Dataa luetaan buf-taulukkoon, joka muutetaan std::stringiksi ja konkatenoidaan result-muuttujaan. Jokaisen luvun jälkeen sisään tulevan datan puskuri tyhjätään tcflush()-funktiolla. Lukusilmukka esitetään esimerkkikoodissa 2.

```
while (bytes_at_port > 0 ||
      (bytes_at_port == 0 && timer < 10))
{
      res = read(fd, &buf, sizeof(buf));
      tcflush(0, TCIFLUSH);
      usleep(20000);
      timer++;
```

```
            buf[res] =  '\0';
            temp = std::string(buf);
            result = result + temp;
            std::memset(&buf, 0,sizeof(buf)) ;
            ioctl(fd, FIONREAD, &bytes_at_port);
            emit UpdateGUI();
    }
```

Esimerkkikoodi 2. Lukusilmukka

Luokan funktiot SerialDevice::Measure() ja SerialDevice::Configure() on jätetty pois tästä dokumentista, koska ne sisältävät lähinnä anturille lähetettäviä käskyjä, joita yritys ei halua antaa vapaaseen levitykseen.

4.3 Kemometrinen malli

MainWindow-luokka sisältää kemometriamallin tietorakenteet, jotka esitetään esimerkkikoodissa 3. Jokainen kemometriamalli on oma struktuurinsa, joka määrittelee mallin aallonpituudet sekä komponentit. Jokainen komponentti vuorostaan on oma struktuurinsa, joka sisältää kyseisen komponentin nimen, b-kertoimet, b-offsetin sekä konsentraatiorajat, joiden sisällä malli on pätevä.

```
struct Component{
    std::string name;
    std::vector<float> bfactors;
    float minresult, maxresult, boffset;
};
struct ChemometricModel
{
    std::vector<float> wavelengths;
    std::vector<Component> components;
};
```

Esimerkkikoodi 3. Kemometriamallin tietorakenteet

MainWindow::LoadChemometricModel() avaa kemometriamallin sisältävän tiedoston ja parsii siitä halutut tiedot. Funktio käsitellään tässä typistettynä, koska siinä on jonkin verran toistoa.

Tiedosto avataan ifstream-objektia käyttäen. Tiedoston nimi on kovakoodattu. Joustavampi ratkaisu olisi ladata tarjolla olevat kemometriamallitiedostot hakemistosta dynaamisesti.

Tekstin parsimisessa hyödynnetään stringstream-objektia. Se toimii käytännössä identtisesti iostream- ja fstream-objektien kanssa. Erityisenä hyötynä on sen automaattinen kokonais- ja liukulukujen parsiminen.

Tiedot ovat tiedostossa sarakkein erotetussa taulukossa. Ensimmäinen sarake sisältää selitetietoa sekä aallonpituudet. Sen jälkeisissä sarakkeissa ovat komponentit, joita voi olla yksi tai useampia. Esimerkkitiedostossa komponentteja on kaksi, eli tiedostossa on sarakkeita yhteensä kolme. Ohjelma kykenee lukemaan tiedostosta mielivaltaisen määrän komponentteja. Parsittavan tiedoston malli esitetään taulukossa 2.

Taulukko 2. Kemometriatiedoston sisältö

Wavelength	Ethanol (vol-%)	Sugar (g/L)
1	0	0
0	50	100
B_0	0	0
1950	-13.89453	-218.184
1945	-30.04533	-416.734
1940	-8.037733	-188.5521
…	…	…

Tiedostoa luetaan rivi kerrallaan. Ensin selvitetään sarakkeiden määrä laskemalla ensimmäisen rivin sarakkeenvaihtomerkit "\t". Sen jälkeen komponenttien nimet luetaan tietorakenteeseen. Koska nimet ovat stringejä, joissa voi olla välilyöntejä, niiden lukemiseen joutuu käyttämään getline()-komentoa, joka on ohjeistettu katkaisemaan syötteen lukeminen sarakkeenvaihtomerkin kohdalla.

Ensimmäisessä sarakkeessa on tiedoston alkupäässä ohjelmalle tarpeetonta selitetietoa. Sen sisältö luetaan garbage-nimiseen muuttujaan ja unohdetaan.

Siinä missä string-tyyppiset tiedot joutuu lukemaan getline()-funktiolla, numeroarvot saa luettua kätevämmin >>-operaattorilla. Operaattori lukee tiedon seuraavaan välilyöntiin, sarakkeen- tai rivinvaihtoon asti. Jos syöte ohjataan esimerkiksi float-tyyppiseen muuttujaan, operaattori yrittää automaattisesti muuntaa lukemaansa tietoa samaan tyyppiin. Ohjelmoijan ei siis tarvitse itse huolehtia tyyppimuunnoksista, mikä on stringstream-objektin käytön etu.

Loppupää tiedostosta sisältää aallonpituudet ja b-kertoimet. Koska rivejä voi olla mielivaltainen määrä, vektorien kokoa ei kannata määrätä etukäteen, vaan tiedot luetaan vektoreihin käyttämällä niiden push_back()-funktiota, joka lisää uuden alkion vektorin loppuun. Tällöin vektorit kasvavat dynaamisesti juuri sen kokoisiksi kuin on tarpeen. Koska kemometriamallin latauskoodi saatetaan suorittaa uudestaan ohjelman suorituksen aikana, vektorit täytyy aina aluksi tyhjentää.

MainWindow::UpdateAnalysis()-funktio laskee absorbanssiarvot kemometristä mallia varten ja laskee niillä lopputuloksen. Laskenta esitetään esimerkkikoodissa 4. Kymmenkantaisen logaritmin saa laskettua log10()-funktiolla, joka löytyy math.h-kirjastosta ja vektorien pistetulon inner_product()-funktiolla, joka löytyy Numeric-kirjastosta. Laskennassa käytettävät vektorit ovat kaikki QVector-tyyppiä.

```
absorbance.resize(wavelength.size());
for (int i = 0; i < adu.size(); i++)
{
    absorbance[i] =
      -log10((adu[i]-dark[i])/reference[i]-dark[i]);
}
float result = std::inner_product(absorbance.begin(),
absorbance.end(), model.components[0].bfactors.begin(), 0);
```

Esimerkkikoodi 4. Absorbanssin laskeminen

Funktio myös päivittää käyttöliittymään laskennan lopputuloksen. Se muutetaan ensin stringstreamin avulla std::stringiksi, joka konvertoidaan C-muotoiseksi stringiksi, josta muodostetaan QString, joka lopulta kelpaa Qt:n label-objektille.

4.4 Qt-käyttöliittymä

Käyttöliittymä toteutettiin käyttämällä Qt Creatorin lomake-editoria. Editori mahdollistaa WYSIWYG (What You See Is What You Get) -tyyppisen drag & drop -käyttöliittymäsuunnittelun. Lomakkeelle sijoitetaan käyttöliittymän elementit halutulla tavalla. Niiden sijainnin ja koon voi määritellä absoluuttisen tarkasti, mutta Qt on omimmillaan, kun käytetään layout-määrittelyjä.

Layoutit huolehtivat sisältämiensä elementtien paikasta ja koosta dynaamisesti. Jos ikkunan kokoa muutetaan, layout mukauttaa elementit uuteen kokoon sopivaksi. Samoin käy, jos joitakin elementtejä piilotetaan ajon aikana pois näkyvistä.

Työssä käytettiin kolmea widgettiä käyttöliittymän toteutukseen. Qlabel on tekstiä sisältävä elementti, jonka teksti ei ole ajon aikana ohjelman käyttäjän muutettavissa. Se voi sisältää myös kuvan. QpushButton on standardi painonappi. Painettaessa nappi lähettää clicked()-signaalin.

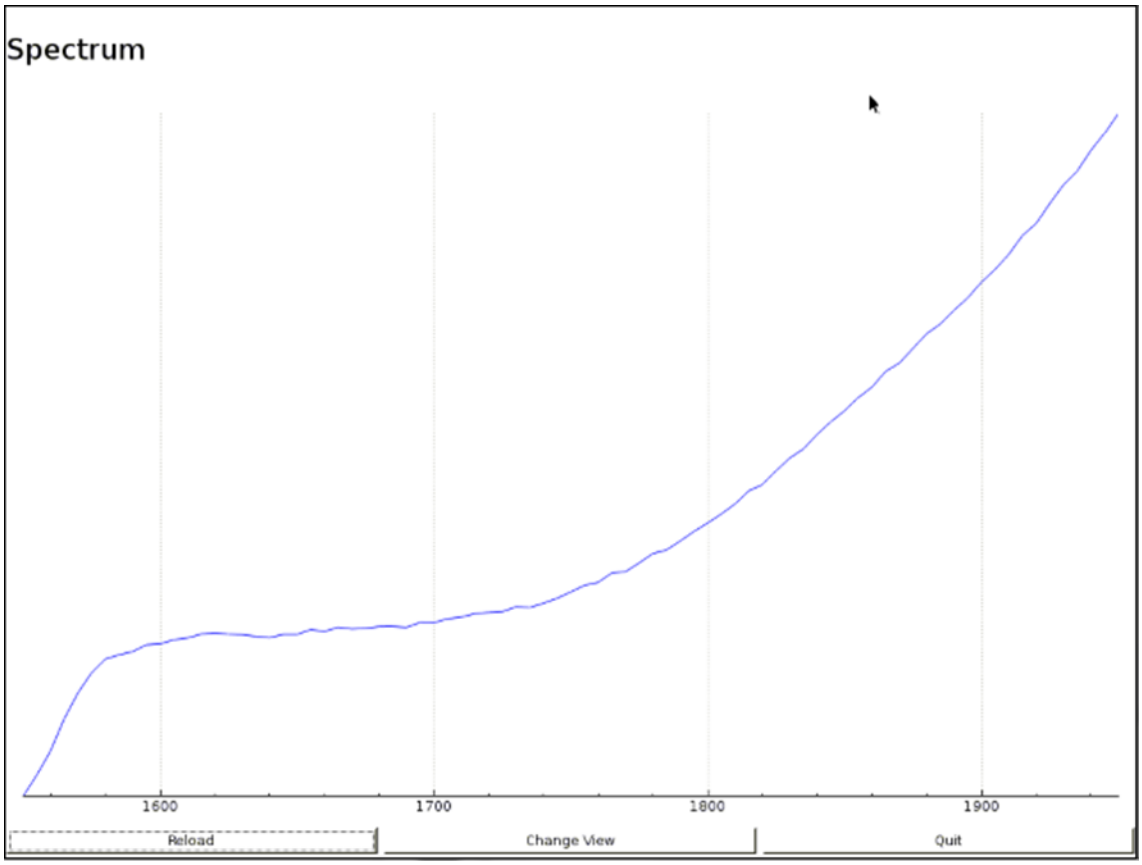
QStackedWidget on hieman abstraktimpi widget. Se sisältää yhden tai useampia sivuja, joista se näyttää kerrallaan vain yhden. Sivut on numeroitu juoksevasti nollasta lähtien. Sen ansiosta samaan ikkunaan voi sisällyttää useamman näkymän, eikä useammalle ikkunalle ole välttämättä tarvetta. QStackedWidgetin toiminnallisuudesta oli erityistä hyötyä tässä projektissa, koska ohjelmaan oli tarkoitus sisällyttää useampia näkymiä ja useamman ikkunan käyttöliittymää olisi ollut hankalaa käyttää pienellä kosketusnäytöllä.

QStackedWidgetillä on Qt Creatorissa erikoinen ominaisuus. Kun Qt Creatorilla tehty ohjelma käynnistetään, ohjelman QStackedWidgetit näyttävät ensimmäiseksi sitä sivua, joka oli Qt Creatorin editorissa auki silloin, kun ohjelma käännettiin. Se ei siis näytä oletuksena ensimmäiseksi esimerkiksi sivua numero nolla, ellei kyseinen sivu ollut viimeisenä auki editorissa.

Ohjelmassa on kaksi päänäkymää: spektrinäkymä ja analyysinäkymä. Näkymät ovat QStackedWidgetin sisällä. Alalaidan kolme nappia Reload, Change View ja Quit ovat QStackedWidgetin ulkopuolella, joten ne ovat yhteiset molemmille näkymille. Reload-nappi lataa kemometriamallin tiedostosta, Change View -nappi vaihtaa näkymää ja Quit-nappi lopettaa ohjelman.

4.4.1 Spektrinäkymä

Spektrinäkymässä on yksi Qlabel-objekti sekä QCustomPlot-objekti. QCustomPlot-objekti näyttää mitatun spektrin ja automaattisesti skaalaa sekä vaaka- että pystyasteikon oikean kokoiseksi. Pystyasteikko on piilotettu, jotta itse käyrällä olisi enemmän tilaa. Näkymä esitetään kuvassa 13.



Kuva 13. Spektrinäkymä

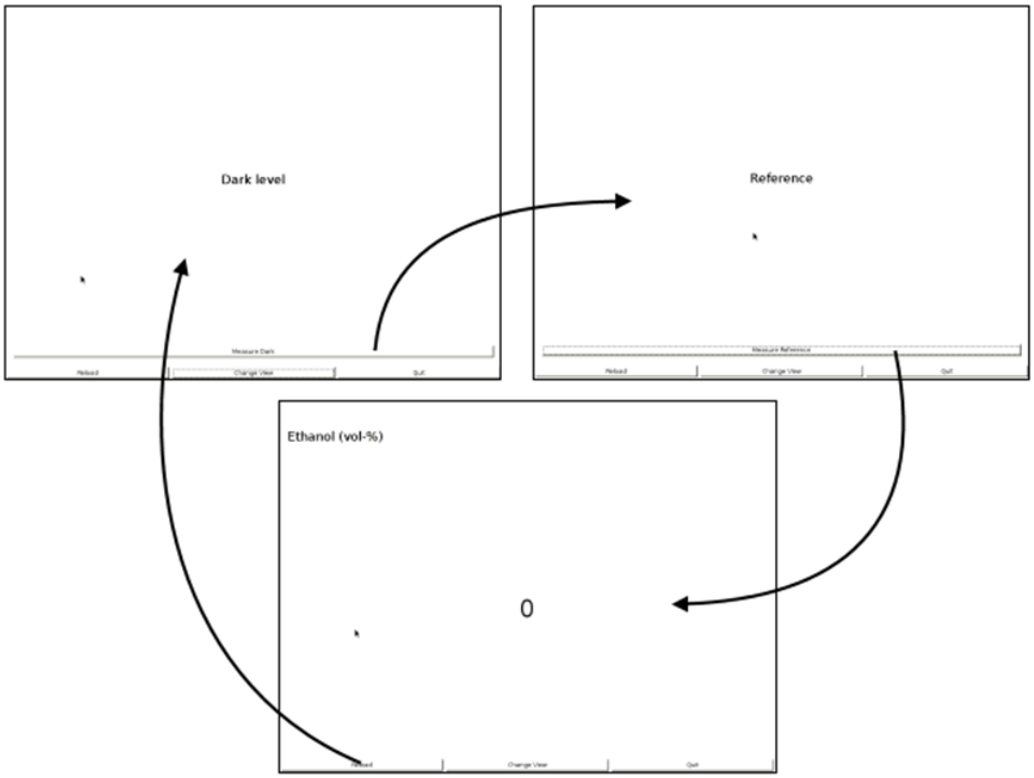
Haittapuolena pystyakselin piilottamisessa on se, että signaalin voimakkuutta ei pysty lukemaan sen asteikolta, vaan sen voi korkeintaan arvioida siitä, kuinka näkyvää signaalin kohina on.

4.4.2 Analyysinäkymä

Analyysinäkymässä on yksi QStackedWidget-objekti, jossa on kolme sivua. Syy ratkaisuun on se, että kemometriamallin käyttö vaatii anturin pimeäsignaalin ja lampun signaalin määrittämistä.

QStackedWidgetin ensimmäisellä sivulla pyydetään mittaamaan pimeäsignaali. Käytännössä ohjelma vain suorittaa normaalin mittauksen. Valonlähteen pimentäminen jää ohjelman käyttäjän tehtäväksi. Measure-napin painaminen vaihtaa sivua.

Toisella sivulla pyydetään mittaamaan referenssitaso. Jälleen kerran ohjelma vain suorittaa mittauksen ja käyttäjän tulee huolehtia siitä, että valonlähde on päällä ja kyvetti tyhjä, ennen kuin painaa Measure-nappia. Napin painaminen vaihtaa sivua.



Kuva 14. Analyysinäkymän silmukka

Kolmannella ja viimeisellä sivulla näkyy kemometriamallin perusteella laskettu konsentraatio kyvetissä olevalle näytteelle. Mikäli Reload-nappia painetaan, kemometriamalli ladataan uudelleen tiedostosta. Samalla QStackedWidgetin sivu vaihdetaan takaisin nollaksi, jolloin pimeäsignaalin ja referenssitason määritys tehdään uudestaan. Tämä voi olla tarpeellista jos esimerkiksi vaihdetaan kyvettiä. Analyysinäkymä siis kiertää kehää, joka on esitetty kuvassa 14.

4.5 Qt:n liittäminen muuhun ohjelmakoodiin

MainWindow-luokka toimittaa tietyssä mielessä pääluokan virkaa. Sen konstruktoria kutsutaan ohjelman main-funktiossa ja asiat, mitkä normaalisti tehtäisiin main-funktiossa, tehdään tämän Qt-ohjelman tapauksessa tässä konstruktorissa. MainWindow-luokka liittää yhteen Qt:n käyttöliittymän ja muun C++-koodin. Se luo muista luokista objektit omaan käyttöönsä. Se ei siis itsessään ole monikäyttöinen ja modulaarinen, vaan sisältää ohjelman sovelluskohtaiset kentät ja funktiot.

MainWindow-luokka perii QMainWindow-luokan, joka taas perii QObject-luokan. Signaali- ja slottijärjestelmän saa tällöin käyttöön QOBJECT-makrolla.

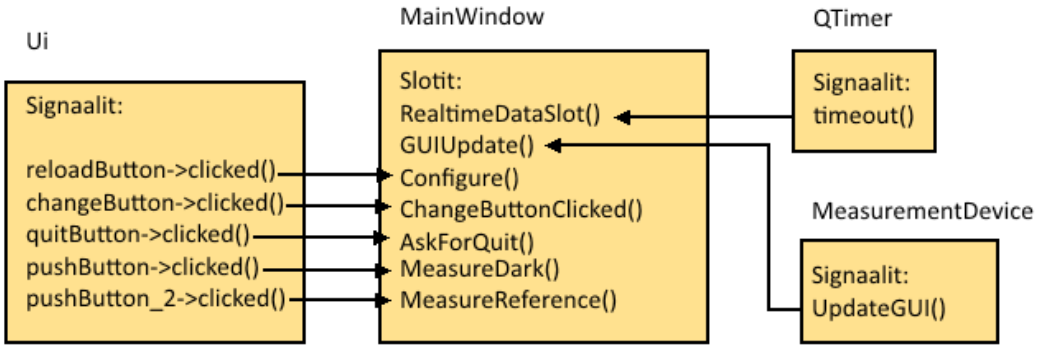
MainWindow-luokan konstruktori suoritetaan kerran ohjelman käynnistyessä. Ensin käyttöliittymä ladataan. Sitten luodaan ajastin, joka kutsuu määräväliajoin slottia, joka lähettää anturille mittauskäskyn.

Ohjelmassa määritellään anturille oletusasetukset siltä varalta, että niitä ei saada luettua tiedostosta. Anturi näkyy käyttöjärjestelmälle aina osoitteessa "/dev/ttyACM0", joten tämä arvo on mahdollista kovakoodata. Joustavampi vaihtoehto olisi kuitenkin tutkia /dev-kansiota ja mahdollisesti jopa luoda lista antureista, jos laitteessa on niitä kiinni useampia.

Seuraavaksi sarjaporttia käyttävä anturi alustetaan, luetaan mittausasetukset tiedostosta ja lähetetään asetukset anturille.

QCustomPlot-objekti alustetaan lisäämällä siihen kuvaaja ja asettamalla x-akseli vastaamaan aallonpituusaluetta. Koska näyttö on niin pienikokoinen, x-akselin asteikkoa harvennetaan ja y-akseli piilotetaan kokonaan.

Signaalit liitetään slotteihin konstruktorin lopussa. Signaalien kytkeytyminen slotteihin esitetään kuvassa 15. Suurin osa ohjelman signaaleista koskee nappien painalluksia. Napin painaminen lähettää clicked()-signaalin, joka aiheuttaa halutun slotin suorittami-



sen.

Kuva 15. Signaalien kytkeytyminen slotteihin

Konstruktorissa aikaisemmin luotu ajastin kytketään timeout()-signaalinsa kautta slottiin RealtimeDataSlot(). Toinen ehkä hieman epätavallisempi signaali on UpdateGUI(), jota kutsumalla saadaan käyttöliittymä pysymään vastaanottavana myös pitkäaikaisen prosessoinnin aikana.

Lopuksi ajastin käynnistetään. Ajastin ottaa start()-funktionsa parametrina hälytysvälin millisekunteina. Ohjelman tapauksessa ajastin kutsuu hälytykseensä kytkettyä slottia yhden millisekunnin välein. Slottia ei kuitenkaan ajeta välittömästi hälytyksen jälkeen, vaan se menee suoritusjonon hännille. Ajastimen ei pitäisi siis häiritä muiden slottien toimintaa keskeyttelemällä niiden suoritusta.

Ohjelman destruktorissa tuhotaan hyvän ohjelmointitavan mukaisesti delete-komennolla dynaamisesti allokoitu muisti.

Funktiota MainWindow::RealtimeDataSlot() kutsutaan aina kun ajastin hälyttää. Se lukee anturilta uudet mittatiedot, tekee niille kemometrisen analyysin ja syöttää ne QCustomPlotin kuvaajaan.

MainWindow::GUIUpdate()-funktiota kutsutaan pitkäaikaisen prosessoinnin aikana. Tämä toiminnallisuus kompensoi sitä, että käyttöliittymä ja sarjaporttikommunikaatio eivät ole omissa, erillissä säikeissään.

5 Yhteenveto

Työn tarkoituksena oli rakentaa Linux-pohjainen ohjelmisto ohjaamaan spektrometrianturia. Lopputulos on ohjelmisto, joka kommunikoi anturin kanssa ja näyttää tiedon näyttöruudulla. Spektri päivittyy uuteen noin sekunnin välein. Ohjelma kykenee laskemaan spektristä absorbanssiarvot ja niistä kemometrisen mallin avulla konsentraation. Näiltä osin työssä asetetut tavoitteet siis saavutettiin.

Ohjelmassa olisi tarvetta moniajolle. Sekä mittaus että käyttöliittymä kilpailevat jatkuvasti keskenään järjestelmän resursseista. Usein tällaisissa tapauksissa nämä kaksi toimintoa erotetaan omiin säikeisiinsä, mutta tässä ohjelmassa ongelma on kierretty antamalla ”etuajo-oikeus” mittaukselle ja pyytämällä aina välillä mittauksen lomassa käyttöliittymää päivittämään itsensä.

Järjestelmän määrittely on vielä hieman kesken. Tarvitseeko mittausasetuksia tai kemometrista mallia kyetä muuttamaan ohjelman ajon aikana? Montako eri komponenttia sen tulee kyetä mittaamaan samanaikaisesti? Mitä kaikkea järjestelmällä tulee voida tehdä kosketusnäytön avulla? Entä esimerkiksi etäyhteyden kautta?

Iso kysymys on myös se, miten mittausasetuksia tai kemometrisia malleja syötetään ohjelmalle. Tällä hetkellä ne ovat kovakoodattuja. Tulisiko ohjelman esimerkiksi tutkia hakemistonsa sisältö ja poimia sieltä tietyillä kriteereillä kemometriamalleja sisältävät tiedostot? Raspberry Pi:llä voi myös pyörittää web-serveriä. Voisiko tiedot syöttää sille esimerkiksi jonkinlaisen web-selaimella täytettävän lomakkeen kautta?

Käyttöliittymän ulkoasu ja sisältö tuskin ovat myöskään lopullisia. Reload-nappi on käytännössä lähinnä debuggauskäyttöön tarkoitettu, eivätkä napit välttämättä edes ole geometrialtaan sormille sopivia.

Järjestelmän kommunikaatio anturin kanssa on siis melko pitkälle määritetty, mutta järjestelmän käyttäjärajapinta on vielä hieman keskeneräinen. Kaikki peruspalikat ovat kuitenkin jo paikoillaan, joten käyttöliittymän hiomisen olettaisi olevan melko kivuton prosessi.

Lähteet

1 Hollas, J. Michael. 2004. Modern Spectroscopy. 4th ed. Chichester: Wiley.

2 Rogalski, Antonio. 2002. Infrared detectors: an overview. Infrared Physics & Technology. Vol 43, s. 187-210.

3 Hecht, Eugene. 2003. Optics. 4th ed. San Francisco: Addison-Wesley.

4 Ikonen, Erkki. 2006. Optiikan perusteet. Espoo: TKK.

5 Maluf, Nadim. 2000. An introduction to microelectromechanical systems engineering. Boston: Artech House.

6 Antila, Jarkko, Tuohiniemi, Mikko, Rissanen, Anna, Kantojärvi, Uula, Lahti, Markku, Viherkanto, Kai, Kaarre, Marko, Malinen, Jouko. 2014. MEMS- and MOEMS-Based Near-Infrared Spectrometers. Encyclopedia of Analytical Chemistry, s. 1–36.

7 Leem, Jung, Woo, Guan, Xiang-Yu, Yu, Jae, Su. 2014. Tunable distributed Bragg reflectors with wide-angle and broadband high-reflectivity using nanoporous/dense titanium dioxide film stacks for visible wavelength applications. Optics Express. Vol 22, s. 18519-18526.

8 Gemperline, Paul. 2006. Practical Guide To Chemometrics, 2nd ed. Boca Raton: CRC Press.

9 Wiberg, Kent. 2004. Multivariate spectroscopic methods for the analysis of solutions. Väitöskirja. Tukholma: Tukholman yliopisto.

10 About Us. Verkkodokumentti. <https://www.raspberrypi.org/about/>. Luettu 30.4.2015

11 Raspberry Pi Hardware. Verkkodokumentti. <https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md>. Luettu 30.4.2015

12 Raspbian FAQ. Verkkodokumentti. <http://www.raspbian.org/RaspbianFAQ>. Luettu 30.4.2015

13 Adafruit PiTFT - 2.8" Touchscreen Display for Raspberry Pi. Verkkodocumentti. <https://learn.adafruit.com/adafruit-pitft-28-inch-resistive-touchscreen-display-raspberry-pi>. Luettu 30.4.2015

14 Vis, Peter. Raspberry Pi LCD DSI Display Connector. Verkkodokumentti. <http://www.petervis.com/Raspberry_PI/Raspberry_Pi_LCD/Raspberry_Pi_LCD_DSI_Display_Connector.html> Luettu 30.4.2015

15 PiTFT: boot to desktop problem. Verkkodokumentti. <https://www.raspberrypi.org/forums/viewtopic.php?p=485302>. Luettu 30.4.2015

16 Sweet, Michael R.Serial Programming Guide for POSIX Operating Systems. Verkkodokumentti. <https://www.cmrr.umn.edu/~strupp/serial.html>. Luettu 30.4.2015

17 Tardieu, Samuel. 2013. What is the difference between /dev/ttyUSB and /dev/ttyACM? Verkkodokumentti. <https://www.rfc1149.net/blog/2013/03/05/what-is-the-difference-between-devttyusbx-and-devttyacmx/>. Luettu 30.4.2015

18 Frerking, Gary. 2001. Serial Programming HOWTO. Verkkodokumentti. <http://tldp.org/HOWTO/Serial-Programming-HOWTO/>. Luettu 30.4.2015

19 Blanchette, Jasmin, Summerfield, Mark. 2006. C++ GUI Programming with Qt 4. 2nd ed. Stoughton: Prentice Hall.

20 Qt (software). Verkkodokumentti. <http://en.wikipedia.org/wiki/Qt_(software>. Luettu 30.4.2015

21 QObject Class. Verkkodokumentti. <http://doc.qt.io/qt-5/qobject.html>. Luettu 30.4.2015

22 Using the Meta-Object Compiler (moc). Verkkodokumentti. <http://doc.qt.io/qt-5/moc.html>. Luettu 30.4.2015

23 Signals & Slots. Verkkodokumentti. <http://doc.qt.io/qt-5/signalsandslots.html>. Luettu 30.4.215

24 Any plan for pie chart. Verkkodokumentti. <http://www.qcustomplot.com/index.php/support/forum/224>. Luettu 30.4.2015

25 Setting up QcustomPlot. Verkkodokumentti. <http://www.qcustomplot.com/index.php/tutorials/settingup>. Luettu 30.4.2015

26 Basics of plotting with QcustomPlot. Verkkodokumentti. <http://www.qcustomplot.com/index.php/tutorials/basicplotting>. Luettu 30.4.2015

27 QMainWindow Class. Verkkodokumentti. <http://doc.qt.io/qt-4.8/qmainwindow.html>. Luettu 30.4.2015

28 QApplication Class. Verkkodokumentti. <http://doc.qt.io/qt-4.8/qapplication.html>. Luettu 30.4.2015

Lähdekoodi

main.cpp:

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowState(w.windowState() ^ Qt::WindowFullScreen);
    w.show();
    return a.exec();
}
```

Mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "SerialDevice.h"

#include <QMainWindow>
#include <QTimer>
#include <QString>
#include <iostream>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <numeric>
#include <math.h>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    //Chemometric structures
    struct Component{
        std::string name;
        std::vector<float> bfactors;
        float minresult, maxresult, boffset;
    };

    struct ChemometricModel
    {
        std::vector<float> wavelengths;
        std::vector<Component> components;
    };

    //Fields related to the UI
```

```
    Ui::MainWindow* ui;
    QTimer* dataTimer;
    int page;

    //Fields related to QCustomPlot
    QVector<double> wavelength;
    QVector<double> adu;

    //Fields related to the serial device
    SerialDevice device;
    std::string device_name;
    float maxwl;
    float minwl;
    int points;
    float spacing;
    int averaging;
    std::string command;

    //Fields related to file I/O
    std::string previousdevice;
    int previouspoints;
    int previousaveraging;

    //Fields related to chemometrics
    ChemometricModel model;
    QVector<double> absorbance,dark,reference;
    bool validanalysis;

    //Calculates concentration from data
    void UpdateAnalysis();

private slots:
    //Gets data from sensor and updates graph
    void RealtimeDataSlot();
    //Processes GUI events
    void GUIUpdate();
    //Loads new configuration from file
    void Configure();
    //Changes the stackedwidget page
    void ChangeButtonClicked();
    //Enables reload button
    void AskForQuit();
    //Closes window
    void ActuallyQuit();
    //Load chemometric model from file
    void LoadChemometricModel();
    //Measure dark and reference signal levels
    void MeasureDark();
    void MeasureReference();
};
#endif
```

MainWindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{

    //Load the user interface
    ui->setupUi(this);
```

```
    //Create timer
    dataTimer = new QTimer(this);

    //Initialize default values
    device_name = previousdevice = "/dev/ttyACM0";
    points = previouspoints = 256;
    averaging = previousaveraging = 5;
    page = 0;
    validanalysis = false;

    //Initialize serial device
    device.AssignDevice(device_name);
    device.Initialize();

    //Load measurement settings from file
    Configure();

    //Send the averaging and wavelengths to the serial device
    device.Configure(averaging, wavelength.data(),
                     wavelength.size());

    //Setup the graph
    ui->customPlot->addGraph();
    ui->customPlot->xAxis->setRange(minwl, maxwl);
    ui->customPlot->xAxis->setAutoTickCount(3);
    ui->customPlot->yAxis->setVisible(false);

    //Connect signals and slots associated with MainWindow

    //Used to run the measurement constantly
    connect(dataTimer, SIGNAL(timeout()), this, SLOT(RealtimeDataSlot()));
    //Pressing Quit button tells the program is about to quit
    connect(ui->quitButton, SIGNAL(clicked()), this, SLOT(AskForQuit()));
    //Reload button reloads measurement setting from file
    connect(ui->reloadButton, SIGNAL(clicked()), this, SLOT(Configure()));
    //Toggles the UI view
    connect(ui->changeButton, SIGNAL(clicked()), this,
SLOT(ChangeButtonClicked()));
    //Called to handle UI events during serial communications processing,
//lessens UI freezing
    connect(&device, SIGNAL(UpdateGUI()), this, SLOT(GUIUpdate()));
    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(MeasureDark()));
    connect(ui->pushButton_2, SIGNAL(clicked()), this,
SLOT(MeasureReference()));

    //Start the timer every 0 milliseconds i.e. constantly
    dataTimer->start(1);
}

MainWindow::~MainWindow()
{
     //ui object is created with "new" so it needs to be deleted manually
    delete ui;
    delete dataTimer;
}

void MainWindow::RealtimeDataSlot()
{
     //If Reload button has been pressed, reload new settings before running
     //measurement
    if(device.GetCalibrationStatus())
        Configure();

    //If measurement isn't already running, run measurement
```

```
    if (!device.GetMeasurementStatus())
    {
        //If Quit button has been pressed, quit instead of running measurement
        if(device.WantToQuit())
        {
            ActuallyQuit();

        }

        //Read new measurement data to adu
        device.Measure(adu.data());

        //Update the graph with the new data
        ui->customPlot->graph(0)->setData(wavelength, adu);
        ui->customPlot->rescaleAxes();
        ui->customPlot->replot();

        //Update the chemometric analysis of the spectrum
        if(validanalysis)
        {
            UpdateAnalysis();
        }
    }
}

void MainWindow::Configure()
{
    //Stop the timer that is constantly running measurement
    dataTimer->stop();
    emit ReloadDisable();
    //If measurement is currently running, just raise a flag otherwise reload
//chemometric model from file
    if (device.GetMeasurementStatus())
        device.SetCalibrationStatus(true);
    else
    {
        validanalysis = 0;
        LoadChemometricModel();
        wavelength.resize(model.wavelengths.size());
        adu.resize(model.components[0].bfactors.size());

        for (int i = 0; i < wavelength.size(); i++)
        {
            wavelength[i] = model.wavelengths[i];
        }
        //Force user to measure dark and reference levels again
        ui->stackedWidget_2->setCurrentIndex(0);

        device.SetCalibrationStatus(false);
    }
    //Enable Reload button
    emit ReloadEnable();

    //Start timer again
    dataTimer->start(1);

}

void MainWindow::AskForQuit()
{
    device.WarnAboutQuit();
}

void MainWindow::ActuallyQuit()
{
```

```
    this->close();
}

void MainWindow::ChangeButtonClicked()
{
    //Cycle through pages 0 and 1
    page = (page + 1) % 2;
    ui->stackedWidget->setCurrentIndex(page);
}

void MainWindow::GUIUpdate()
{
    //Process any pending events coming from the UI
    QCoreApplication::processEvents();
}

void MainWindow::EnableReload()
{
    //Enable Reload button
    ui->reloadButton->setEnabled(true);
}

void MainWindow::DisableReload()
{
    //Disable Reload button
    ui->reloadButton->setEnabled(false);
}

void MainWindow::UpdateAnalysis()
{
    //Calculate absorbance
    absorbance.resize(wavelength.size());

    for (int i = 0; i < adu.size(); i++)
    {
        absorbance[i] = -log10((adu[i]-dark[i])/reference[i]-dark[i]);
    }

    float result = std::inner_product(absorbance.begin(), absorbance.end(),
model.components[0].bfactors.begin(), 0);
    std::stringstream ss;
    ss << result << " %";
    std::string temp;
    ss>> temp;
    QString labeltext(temp.c_str());
    ui->label_3->setText(labeltext);
}

void MainWindow::LoadChemometricModel()
{
    std::ifstream file;

    std::string temp, temp2;
    std::string garbage;
    float temporary;
    int componentnr = 0;

    file.open("ANALYSIS_water-ethanol_50prcnt_sugar-100gpL");

    if(file.is_open())
    {
        //Read component names
        std::stringstream ss, ss2;
        getline(file, temp);
        ss << temp;
```

```
        componentnr = std::count(temp.begin(), temp.end(), '\t');
        model.components.resize(componentnr);
        getline(ss, garbage,'\t');
        for (int i = 0; i < componentnr; i++)
        {
            getline(ss, model.components[i].name,'\t');
        }

        //Read minimum valid result
        getline(file, temp);
        ss.clear();
        ss << temp;
        ss >> garbage;
        for (int i = 0; i < componentnr; i++)
        {
            ss >> model.components[i].minresult;
        }

        //Read maximum valid result
        getline(file, temp);
        ss.clear();
        ss << temp;
        ss >> garbage;
        for (int i = 0; i < componentnr; i++)
        {
            ss >> model.components[i].maxresult;
        }

        //Read b-offset
        getline(file, temp);
        ss.clear();
        ss << temp;
        ss >> garbage;
        for (int i = 0; i < componentnr; i++)
        {
            getline(ss, temp2, '\t');
            ss2.str(temp2);
            ss2 >> model.components[i].boffset;
            model.components[i].bfactors.clear();
        }
        model.wavelengths.clear();

        //Read wavelengths and b-factors
        while (getline(file, temp))
        {
            ss.clear();
            ss << temp;
            ss >> temporary;
            model.wavelengths.push_back(temporary);
            for (int i = 0; i < componentnr; i++)
            {
                ss >> temporary;
                model.components[i].bfactors.push_back(temporary);
            }
        }
        std::string temp = model.components[0].name;
        QString labeltext(temp.c_str());
        ui->label->setText(labeltext);
    }

    file.close();
}

void MainWindow::MeasureDark()
{
```

```
    dark.resize(wavelength.size());
    device.Measure(dark.data());
    ui->stackedWidget_2->setCurrentIndex(1);
}

void MainWindow::MeasureReference()
{
    reference.resize(wavelength.size());
    device.Measure(reference.data());
    ui->stackedWidget_2->setCurrentIndex(2);
    validanalysis = true;
}
```

MeasurementDevice.h:

```
#ifndef MEASUREMENTDEVICE_H
#define MEASUREMENTDEVICE_H

#include <vector>
#include <string>
#include <iostream>
#include <sstream>

#include <QObject>



class MeasurementDevice: public QObject
{
    Q_OBJECT

    public:
    MeasurementDevice();
    //Getters
    int GetMinwl();
    int GetMaxwl();
    bool GetCalibrationStatus();
    bool GetMeasurementStatus();
    //Setters
    void SetCalibrationStatus(bool);
    //Called to let sensor know that quit is imminent
    void WarnAboutQuit();
    //Returns whether or not the sensor thinks the program is about to quit
    bool WantToQuit();

            protected:
    //Structure holding information about the sensor
    struct DeviceInformation {
        std::string type;
        int version;
        int serialnb;
        int mandate;
        int caldate;
        std::vector<double> coeffs;
        int minwl;
        int maxwl;
        int maxcount;
        int res25;
        int B;
        int tempsetpoint;
    };

    //Structure containing the settings used for measurement
    struct MeasurementSettings
```

```
    {
        int averaging;
        float wavelengths[512];
        int points;
    };

        //Constructs the sensor information structure from provided string
                        int BuildDeviceInfo(std::string);
        struct DeviceInformation info;
        struct MeasurementSettings settings;
        //Flags for process control
        bool calibration_on, measurement_on, going_to_quit;
        //Latest measurement data
        float data[512];

    public slots:
        //Fills the data array with new data
        virtual int Measure(double* results) = 0;
        //Sends new measurement settings to device
        virtual int Configure(int averaging, double* wls, int numberofwls) =
0;
        //Initializes the communication path and fills the DeviceInformation
structure
                        virtual int Initialize() = 0;


    signals:
        //Asks program to process GUI events
        void UpdateGUI();
};

#endif
```

MeasurementDevice.cpp:

```
#include "MeasurementDevice.h"

int MeasurementDevice::GetMinwl()
{
    return info.minwl;
}

int MeasurementDevice::GetMaxwl()
{
    return info.maxwl;
}

int MeasurementDevice::BuildDeviceInfo(std::string buffer)
{
            std::string temp, keyword;
    unsigned int beginning, pos;

    temp.assign(buffer);

    //Search for values based on keyword, the format expected is "<Keyword>:
<Value>\n"

    keyword = "minWl";
    pos = temp.find(keyword);
    if(pos != std::string::npos)
    {
        beginning = pos + keyword.length() + 2;
        pos = temp.find("\n", beginning);
        if(pos != std::string::npos)
            std::istringstream(temp.substr(beginning ,pos - beginning)) >>
info.minwl;
```

```
    }
    keyword = "maxWl";

    pos = temp.find(keyword);
    if(pos != std::string::npos)
    {
        beginning = pos + keyword.length() + 2;
        pos = temp.find("\n", beginning);
        if(pos != std::string::npos)
            std::istringstream(temp.substr(beginning ,pos - beginning)) >>
info.maxwl;
    }
    return 0;
}

bool MeasurementDevice::GetCalibrationStatus()
{
    return calibration_on;
}

void MeasurementDevice::SetCalibrationStatus(bool setting)
{
    calibration_on = setting;
}

bool MeasurementDevice::GetMeasurementStatus()
{
    return measurement_on;
}

void MeasurementDevice::WarnAboutQuit()
{
    going_to_quit = true;
}

bool MeasurementDevice::WantToQuit()
{
    return going_to_quit;
}

MeasurementDevice::MeasurementDevice()
{
     calibration_on = measurement_on = going_to_quit = false;
     settings = MeasurementSettings();
}
```

SerialDevice.h:

```
#ifndef SERIALDEVICE_H
#define SERIALDEVICE_H

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>

#include <stdio.h>
#include <string>
#include <cstring>

#include <iostream>
#include <fstream>
#include <sstream>
```

```
#include <stdexcept>

#include "MeasurementDevice.h"


#define BAUDRATE B115200

class SerialDevice: public MeasurementDevice
{
    private:
        //fd is the file descriptor pointing at the device
        //res is the number of bytes read
        int fd, res;
        //Buffer for reading ascii input
                        char buf[2048];
        //Buffer for reading binary data
        float databuf[512];
        //devaddress contains the location of the serial port
        //lastcommand contains the command last sent to the device
                        std::string devaddress, lastcommand;
        //New and old serial port settings
        termios oldtio, newtio;
        //Sends the given command to the device
        int SendData(std::string);
        //Reads data from the device based on lastcommand
        int ReceiveData(std::string&);

    public:
        SerialDevice();
                        SerialDevice(const std::string& device);
        //Sets up the serial port
                        int Initialize();
        //Sends new measurement configuration to the device
        int Configure(int averaging, double* wls, int numberofwls);
        //Measures according to measurement settings
        int Measure(double* results);
        //Sets the location of the serial port
        int AssignDevice(const std::string& device);
        ~SerialDevice();
};

#endif
```

SerialDevice.cpp:

```
#include "SerialDevice.h"

SerialDevice::SerialDevice() { }

int SerialDevice::AssignDevice(const std::string& device)
{
   devaddress = device;

   return 0;
}

SerialDevice::SerialDevice(const std::string& device)
{
            devaddress = device;

    Initialize();
}
```

```
int SerialDevice::SendData(std::string command)
{
    //Save the command so that the read function knows what to expect
    lastcommand = command;
    //The commands end in newline
            command = command + "\n";
    write(fd, command.c_str(),command.size());
    //Wait until command has been sent to serial device
    tcdrain(fd);
            return 0;
}

int SerialDevice::ReceiveData(std::string& result)
{

    result = "";

    std::string temp = "";
    int bytes_at_port = 0;
    int timer = 0;
    int totalres = 0;

    //Check if any bytes to read
    ioctl(fd, FIONREAD, &bytes_at_port);
    //Loop as long as there are bytes to read or until a timer has expired
    while (bytes_at_port > 0 || (bytes_at_port == 0 && timer < 10))
            {
                        res = read(fd, &buf, sizeof(buf));
                        tcflush(0, TCIFLUSH);
                        usleep(20000);
                        timer++;
                        //Null terminate the char array to convert to string
                        buf[res] =  '\0';
                        temp = std::string(buf);
                        result = result + temp;
                        std::memset(&buf, 0,sizeof(buf)) ;
                        ioctl(fd, FIONREAD, &bytes_at_port);
                        emit UpdateGUI();
            }
            return 0;
}

int SerialDevice::Initialize()
{
    //Attempt to open communication with the serial device, exit if unsuccess-
ful
            fd = open(devaddress.c_str(), O_RDWR | O_NOCTTY); //open serial
device
            if (fd < 0)
            {
                        perror(devaddress.c_str());
                        throw std::invalid_argument("Device not found!");
                        return -1;
            }

    tcgetattr(fd, &oldtio);

    //Zero the struct for new settings
    newtio = termios();

    //Set baudrate, character size mask,enable receiver
            newtio.c_cflag = BAUDRATE | CS8 | CREAD;
    //Ignore parity, translate carriage return to newline
            newtio.c_iflag = IGNPAR | ICRNL;
```

```
    //Leave output flags alone
    newtio.c_oflag = 0;
    //Canonical mode off, echo off, echo erase character as backspace off,
signal characters off
            newtio.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
    //Flush data received but not read
    tcflush(fd, TCIFLUSH);
    //Set new serial port settings
            tcsetattr(fd, TCSANOW, &newtio);

    info = DeviceInformation();
    //Attempting to read and build device info
    while(info.minwl == 0)
    {
        std::string result;
        ReceiveData(result);

        //Send the received string to be parsed
        BuildDeviceInfo(result);

    }

    //Zero data array
    std::memset(&data, 0, sizeof(data));

            return 0;

}

SerialDevice::~SerialDevice()
{
    while (close(fd) < 0)
    {
    }
    tcsetattr(fd, TCSANOW, &oldtio);
}
```