Vaonis

Rapport d’activité

Stage du 4 mai 2020 au 5 juin 2020

Olivier KEITH
23/04/2021

Table des matières

Présentation de la société

Le stage a été réalisé pour la société Vaonis, qui est start'up basée à Clapiers (34) en télétravail. Vaonis fabrique et commercialise des télescopes tout-en-un abordable permettant de prendre des photos des objets célestes.

Le projet

Lors de la prise de photos astronomiques, il est nécessaire de faire de longue pose, pour recevoir beaucoup de lumière provenant des objets célestes très peu lumineux. Comme les poses sont longues, il faut compenser la rotation de notre planète, sinon, les étoiles s'étirent très rapidement. C'est pourquoi les instruments de prise de photo astronomique sont motorisés. Cependant, la mécanique, aussi précise soit-elle, n'est jamais fiable. Il est donc nécessaire de corriger l'erreur périodique inhérente à toutes mécanique.

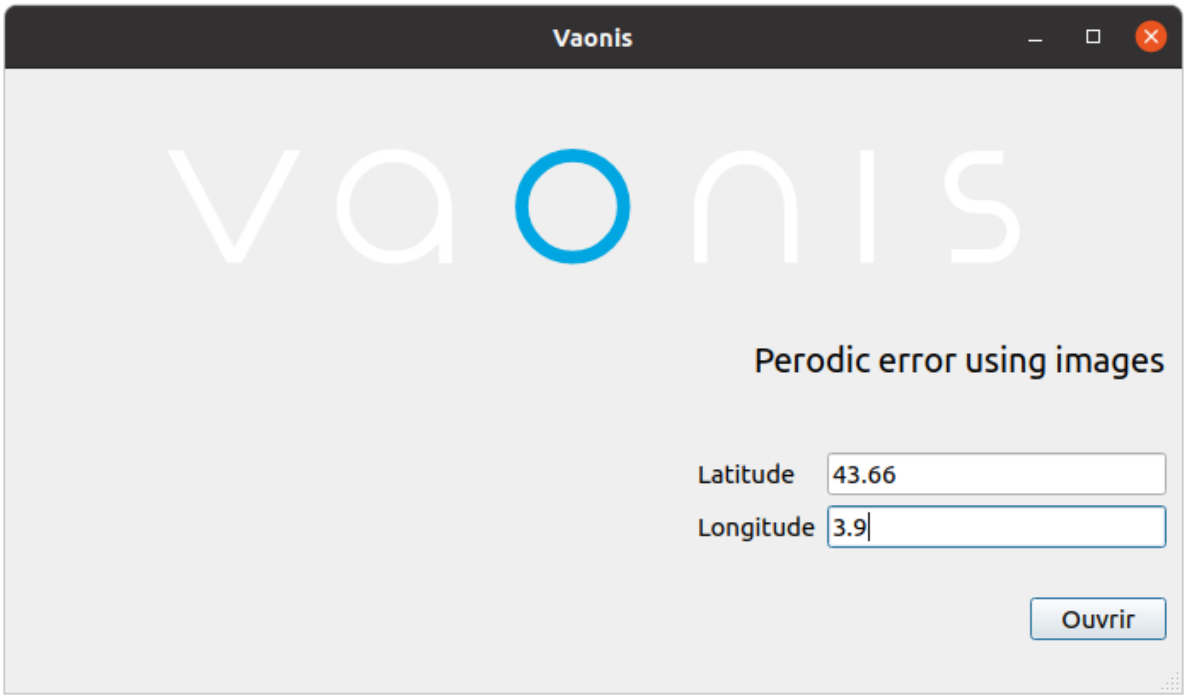
Lors de ce stage, ma mission était de créer entièrement une application permettant de visualiser cette erreur périodique à l'aide de graphiques.

J'ai codé cette application en C++ avec l'aide du Framework Qt sous Linux Ubuntu et selon le modèle MVC.

L'interface de lancement

J'ai tout d'abord mis en place la page de lancement de l'application. Cette fenêtre portant les couleurs de Vaonis dispose de deux champs qu'il faut renseigner de la longitude et la latitude où ont été prises les photos. Puis, en cliquant sur le bouton « ouvrir », l'explorateur se lance afin d'aller chercher l'ensemble de la série de photo dont l'on souhaite connaitre l'erreur périodique.

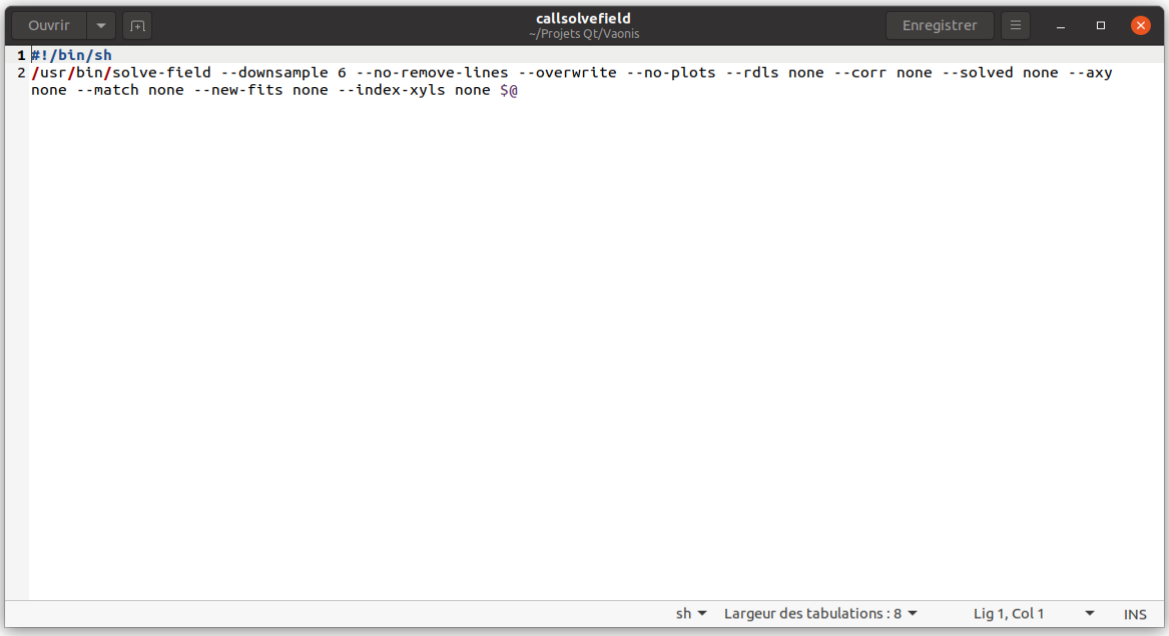
La première chose que doit faire l'application est de déterminer les coordonnées des objets présents dans le champ de l'image, au format équatorial. Cette application existe déjà sous Linux : solve-filed.



Solve-field

Dans un premier temps, il fallait intégrer cette solution qui fonctionne sous Linux à mon application

Je l'ai intégrée grâce à un script shell que j'ai appelé « callsolvefield ». Pour le bon fonctionnement de ce script, il est nécessaire de télécharger et placer les index astrométriques correspondant.



J'ai défini un script par défaut, qu'il est possible de modifier selon ses besoins.

L'appel du script dans le contrôleur à l'aide d'un QString

```
QString commande;
commande = "/home/olivier/Projets Qt/Vaonis/callsolvefield";
```

Le flux

Le programme appelle « solve-field » sur chaque image qui lui retourne les coordonnées équatoriales de chaque objet de chaque image dans un flux unique qu'il a fallu décortiquer. Pour cela, j'ai utilisé les signaux et slots de Qt. Dans la capture suivante, le contrôleur analyse le flux et cherche uniquement les coordonnées (ascension droite (RA) et déclinaison (Dec)) ainsi que la rotation d'angle que l'on a placé dans un tableau.

```
void Controleur::onDataReceived(){

    //Lecture des données envoyées par solve-field
    QByteArray tableau = QByteArray(process_->readAll());

    //traitement des données reçues dès qu'un bloc est entièrement reçu
    process_->readyRead();

    //Bloc des données à traiter
    QString blocDonneeImage(tableau);
    string str_blocDonneeImage = blocDonneeImage.toStdString();

    //déclaration des variables
    int i =0;
    string txt_RADEC = "Field center: (RA,Dec) =";
    string txt_ROT = "Field rotation angle:";
```

Mais, il me manquait la date et l'heure de l'observation, Stellina les enregistre dans l'entête de chaque photo.

Les Fits

Le format utilisé par Stellina pour ses photographies est un format spécifique développé par la NASA : le format Fits.

Le contrôleur doit donc également chercher la date et l'heure dans l'entête de chaque photo.

```
// on parcourt les photos pour trouver les date-heure d'observation
 while (i< (var_cheminImages.size()) && date_heure_obs == ""){

     chemin = var_cheminImages.value(i);
     str_chemin = chemin.toStdString();

     //on exclue de la recherche du chemin la photo déjà traitée et enregistrée dans le vecteur
     if(str_chemin != str_chemin_precedent) {

        //on cherche le chemin de chaque photo sélectionnée dans le bloc de données. On sort de la boucle dès que le chemin est trouvé
        if (str_blocDonneeImage.find(str_chemin) != std::string::npos) {

            //on crée un objet FITS correspondant au fichier de la photo
            std::unique_ptr<FITS> pInfile(new FITS(str_chemin,Read, false));

            //on crée le primary HDU - lorsqu'il existe - du fichier correspondant
            try {

                PHDU& image = pInfile->pHDU();
                //on récupère la date / heure d'observation
                image.readKey("DATE-OBS", date_heure_obs);

                //on gère le cas où on retrouve le chemin de la photo dans le bloc de données alors que le vecteur est déjà créé
                if(date_heure_obs == date_heure_obs_precedent) {
                    date_heure_obs = "";
                }

            } catch (CCfits::FitsException& exept) {

                date_heure_obs = "FitsKey Date/Heure absent";

            }

            //on sort de la boucle si on a trouvé le chemin de la photo dans le bloc traité
            break;
        }

    }
    i++;
}
```

La conversion

Toutes ces données réunies au format « string », il a fallu les extraire et les convertir au format « int ». C'est le contrôleur qui s'en occupe. Je récupère toutes les variables issues de « solve-field » en découpant les chaines stockées dans mon QByteArray.

```
//on vérifie si les coordonnées RA,DEC sont dans le bloc traité
if (str_blocDonneeImage.find(txt_RADEC) != std::string::npos) {

    //On isole dans donneeRADEC la chaine comprise entre "Field center: (RA,Dec) =" et ") deg."
    int index1Debut = str_blocDonneeImage.find("Field center: (RA,Dec) =");
    int index1Fin = str_blocDonneeImage.find(") deg.");
    str_RADEC = str_blocDonneeImage.substr(index1Debut+26, index1Fin-index1Debut-26);

    //On trouve RA et DEC à partir de donneesRADEC
    int index2 = str_RADEC.find(", ");
    str_RA = str_RADEC.substr(0,index2);
    str_DEC = str_RADEC.substr(index2+2,str_RADEC.length());
}


//on vérifie si la coordonnée ROT est dans le bloc traité
if (str_blocDonneeImage.find(txt_ROT) != std::string::npos) {
    //on isole dans donneeROT la chaine comprise entre "Field rotation angle: up is " et " degrees"
    int index3Debut = str_blocDonneeImage.find("Field rotation angle: up is ");
    int index3Fin = str_blocDonneeImage.find(" degrees");
    str_ROT = str_blocDonneeImage.substr(index3Debut+28,index3Fin-index3Debut-28);
}
```

Puis, je sauvegarde chaque point de coordonnée dans un vecteur (QVector), puis, je réinitialise les variables pour créer un nouveau vecteur, et ainsi de suite.

```
 //lorsqu'on a toutes les infos, on crée le vecteur
 if ((str_RA!="") && (str_DEC!="") && (str_ROT!="") && (date_heure_obs !="")) {

     double ra = atof(str_RA.c_str());
     double dec = atof(str_DEC.c_str());
     double rot = atof(str_ROT.c_str());

     CoordEquat sauv_coord(ra, dec, rot, date_heure_obs);
     vecteur_coord.append(sauv_coord);

     //on vide les champs qui ont été renseignés dans le vecteur
     str_RA = "";
     str_DEC = "";
     str_ROT = "";
     str_RADEC = "";
     date_heure_obs_precedent = date_heure_obs ;
     str_chemin_precedent = str_chemin;
     date_heure_obs = "";
     str_chemin = "";
```

Une fois tous les vecteurs créés, le contrôleur procède à la conversion du format équatorial au format alt-azimutale. Je n'affiche dans la capture suivante qu'une partie de cette conversion.

```cpp
void Controleur::conversion(QVector<CoordEquat> vecteur_coord) {

    string item_date_heure;

    //on parcourt le vecteur contenant tous les RA, DEC, Rot et date-heure observation
    for(int i=0; i < vecteur_coord.size(); i++){

        item_date_heure = vecteur_coord.at(i).getDheure();

        //on ne conserve pas les coordonnées lorsqu'on n'a pas la date d'observation
        if (item_date_heure != "FitsKey Date/Heure absent") {
            std::string string_annee = item_date_heure.substr(4,4);
            std::string string_mois = item_date_heure.substr(9,2);
            std::string string_jour = item_date_heure.substr(12,2);
            std::string string_heure = item_date_heure.substr(15,2);
            std::string string_minute = item_date_heure.substr(18,2);
            std::string string_seconde = item_date_heure.substr(21,2);

            double annee = atof(string_annee.c_str());
            double mois = atof(string_mois.c_str());
            double jour = atof(string_jour.c_str());
            double heure = atof(string_heure.c_str());
            double minute = atof(string_minute.c_str());
            double seconde = atof(string_seconde.c_str());


            // formule : JJ = 367*UTC_Y-ENT(7*(UTC_Y+ENT((UTC_MM+9)/12))/4)+ENT(275*UTC_MM/9)
            //                 +UTC_D-730531,5+(UTC_H+UTC_M/60+UTC_S/3600)/24
            double jour_julien = (367*annee)-floor(7*(annee+floor((mois+9)/12))/4)+floor(275*mois/9)+jour-730531.5+(heure+(minute/60)+(seconde/3600))/24;

            //conversion du jour julien en string
            string str_jj = std::to_string(jour_julien);

            // formule gmst = MOD(280,46061837+360,98564736629*JJ+0,000387933*(JJ/36525)^2-((JJ/36525)^3/38710000);360)
            double gmst = fmod((280.46061837+(360.98564736629*jour_julien)+(0.000387933*(jour_julien/36525)*(jour_julien/36525))-((jour_julien/36525)*(jour_julien/36525)*(jour_julien/36525)/38710000)),360);
```

Chaque résultat de conversion est stocké dans un QPoint.

Les graphiques

Tous les QPoints étant créés, il fallait les afficher dans un graphique. J'ai mis en place trois affichages différents que j'ai commenté dans le code.

La plus intéressante affiche les trois droites dans le même graphique.

Le code pour la partie azimutale

```
affichagefinal2::affichagefinal2(QVector<QPointF> vecteur1, QVector<QPointF> vecteur2, QVector<QPointF> vecteur3)
{
    MainWindow *p_mywind = MainWindow::getMainWinPtr();

    //Création des graphes AZIMUT, ALTITUDE et ROTATION
    //Création des QLineSeries
    QLineSeries *series_az = new QLineSeries();
    QLineSeries *series_alt = new QLineSeries();
    QLineSeries *series_rot = new QLineSeries();

    //Ajout des QPoint dans les QLineSeries
    foreach(QPointF point1, vecteur1)
        *series_az<< point1 ;

    foreach(QPointF point2, vecteur2)
        *series_alt<< point2 ;

    foreach(QPointF point3, vecteur3)
        *series_rot<< point3 ;

    //Ajout des légendes
    series_az->setName("Azimut");
    series_alt->setName("Altitude");
    series_rot->setName("Rotation");

    //Gestion des couleurs : AZIMUT en bleu, ALTITUDE en vert et ROTATION en rouge
    QPen pen_az(Qt::blue);
    QPen pen_alt(Qt::green);
    QPen pen_rot(Qt::red);

    //Gestion de l'épaisseur du trait
    pen_az.setWidthF(1.5);
    pen_alt.setWidthF(1.5);
    pen_rot.setWidthF(1.5);
```

```
//on lie les paramètres de Font ci-dessus aux QLineSeries
series_az->setPen(pen_az);
series_alt->setPen(pen_alt);
series_rot->setPen(pen_rot);

//Création du graphe
QChart *chart = new QChart();
chart->legend()->setVisible(true);
//chart->setTitle("Titre du graphe");      ----- A modifier si on veut ajouter un titre au graphique-----

//Pour AZIMUT ------------------------------------------------
//Axe des abscisses
QValueAxis *axe_azX = new QValueAxis;
chart->addAxis(axe_azX, Qt::AlignBottom);

//on force le format des coordonnées en X à 4 décimales
axe_azX->setLabelFormat("%.4f");

//on affiche l'axe des abscisses et ses valeurs en noir
QPen axe_azX_pen(Qt::black);
axe_azX->setLinePen(axe_azX_pen);
axe_azX->setLabelsColor(Qt::black);
//on fixe la taille de la police des valeurs en abscisse
QFont labels_font_azX;
labels_font_azX.setPixelSize(11);
axe_azX->setLabelsFont(labels_font_azX);

//on ajoute un titre à l'axe des abscisses
axe_azX->setTitleText("Jours juliens");
//on fixe la taille de la police et la couleur de ce titre
QFont title_font_azX;
title_font_azX.setPixelSize(12);
axe_azX->setTitleFont(title_font_azX);
QBrush titleX_brush(Qt::black);
axe_azX->setTitleBrush(titleX_brush);

//Axe des ordonnées
QValueAxis *axe_azY = new QValueAxis;
chart->addAxis(axe_azY, Qt::AlignLeft);
```

```
//on force le format des coordonnées en Y à 4 décimales
axe_azY->setLabelFormat("%.4f");

//on affiche les axes des ordonnées et leurs valeurs de la même couleur que la courbe associée
QPen axe_azY_pen(Qt::blue);
axe_azY->setLinePen(axe_azY_pen);
axe_azY->setLabelsColor(Qt::blue);
//on fixe la taille de la police des valeurs en ordonnée
QFont labels_font_azY;
labels_font_azY.setPixelSize(11);
axe_azY->setLabelsFont(labels_font_azY);

//on ajoute la QLineSeries au QChart
chart->addSeries(series_az);

//on lie les axes créés à la QLineSeries
series_az->attachAxis(axe_azX);
series_az->attachAxis(axe_azY);
```

Et le résultat