Rapport de projet NF05

Axel Mousset Aurélien Labate
Université de Technologie de Troyes

Automne 2014

Sommaire

Fichiers source

Chapitre 1

Introduction

Dans le cadre de l'UV NF05, "Introduction au language C", nous avons été amenés à mener à bien un projet de développement logiciel en laguage C (ou dérivé). Le logiciel en question, imposé, est un logiciel graphique de calcul matriciel. Il doit être capable de gérer des matrices de différentes tailles, les contenir sous formes de variables, les faire interagir au travers d'opérations mathématiques simples, et de pouvoir sauvegarder l'état courant du programme. Nous avons décidé de remplir également les objectifs supplémentaires, à savoir l'imbrication des opérations et la résolution d'équations simples.

Dans ce document, nous exposerons les différents outils et processus utilisés, ainsi que les difficultés rencontrées lors de la réalisation de ce projet.
Ainsi, nous verrons dans un premier temps les choix technologiques que nous avons fait, puis les différents algorithmes utilisés pour résoudre les problèmes rencontrés. Enfin, nous présenterons le fonctionnement du programme.
Le code est disponible, en tant qu'annexe, à la fin du document ; il l'est également sur internet [1] : en effet, ce projet se veut libre. Il est ainsi librement accessible et modifiable [2].

1. https ://github.com/ALabate/utt-nf05-project
2. Sous license MIT

Chapitre 2

Choix technologiques

Le choix des technologies utilisée pour le développement du logiciel a été le point d'entrée du projet. Les enjeux sont cruciaux, car la vitesse du développement et la productivité des développeurs dépend du choix des outils qui sont soit déjà connus, soit à apprendre. Le tout a donc été de trouver un accord au sein du binôme sur les technologies à employer. Cette étape de brainstorming a duré environ deux heures.

2.1 Language C++

La première étape dans cette reflexion a été de choisir le language dans lequel il serait développé : la contrainte qui nous était imposée était le choix entre le language C et ses dérivés.
Le C est un language de programmation généraliste et bas niveau [1] inventé dans les années 1970. C'est un des language les plus utilisé au monde, pionnier du monde informatique, qui a inspiré nombre d'autre language et donné naissance à des dérivés comme le C# [2], l'Objective-C ou encore le C++.
Notre choix s'est porté sur ce dernier, car il intègre bon nombre de concepts plus récents en programmation comme la POO [3] qui nous sont familiers. Il n'est également d'aucune affiliation particulière avec les entreprises : le C# étant développé par microsoft et l'objective-C par Apple.
Une fois le language choisi, nous nous sommes intéressés au différents frameworks [4] graphiques disponibles.

2.2 Framework Qt

La motivation derrière l'utilisation de Qt résulte tout d'abord de la volonté d'avoir un programme multi-plateforme. Le développement du programme étant fait dans un environnement linux, et avec une majorité d'ordinateurs tournant sous windows, la nécessité d'avoir un programme portatif était évidente.
Un rapide tour d'horizon sur les différents frameworks/libraries multi-plateformes montre qu'il n'y a essentiellement que trois grand projet : WxWidgets, GTK et Qt.

1. Près de la machine. En C beaucoup de résponsabilités sont laissées au développeur, comme l'allocation mémoire
2. Prononcer "C sharp"
3. Programmation orientée objet
4. Espace de travail - sorte de grande bibliothèque de programmation

FIGURE 2.1 – Logo de Qt

L'avantage de Qt par rapport a ses concurrents est qu'il jouit d'une très large communauté, et est maintenu par Nokia. C'est un framework très moderne allant jusqu'à étendre le C++ pour lui apporter des fonctionnalités non natives. Il permet donc le développement dans un environnement plus haut niveau, au détriment des performances du programme, que nous avons jugées négligeables étant donnée l'échelle du programme.
Enfin, en plus de faciliter le développement du coeur du programme, Qt à l'avantage de proposer un IDE [5], Qt Creator, qui dispose d'une auto-complétion très poussée ainsi qu'un éditeur WYSIWYG [6] d'interface graphique.

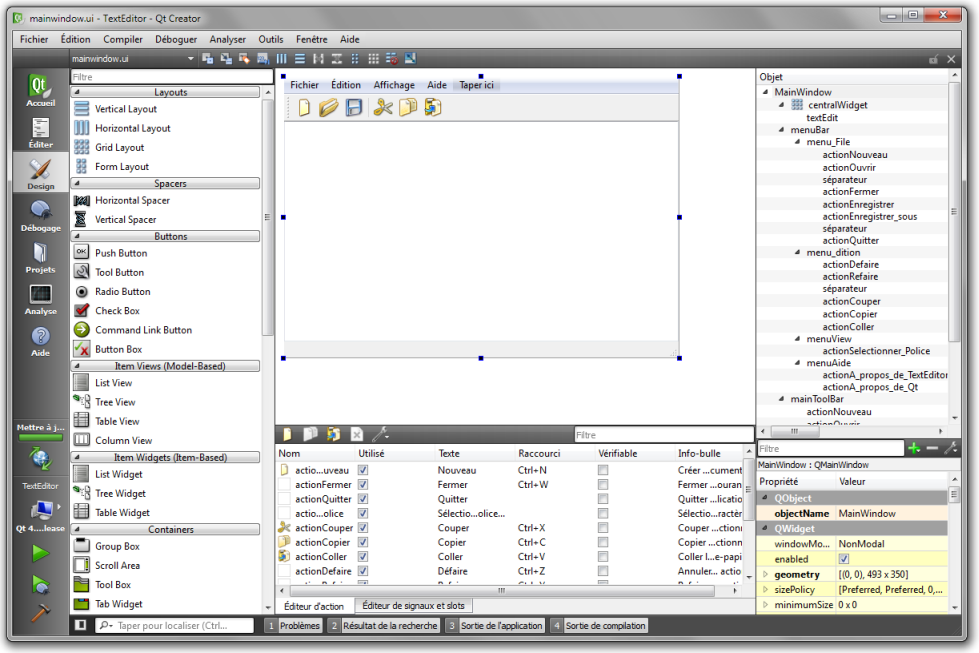

FIGURE 2.2 – Éditeur d'interface WYSIWYG de Qt Creator

L'utilisation de l'éditeur graphique de Qt Creator a fait gagner beaucoup de temps sur le développement. Malgrés sa puissance et sa simplicité d'utilisation, il n'en reste pas moins limité : seul un certain nombre de widget prédéfinis peuvent être utilisés. L'utilisation de widgets plus complexes ont été requis pour la réalisation de la liste des variables, ils ont doncs été développés main (Voir A.5 detailedlist.hpp).

5. Integrated development environment
6. What you see is what you get - il n'y a qu'à dessiner pour obtenir ce que l'ont veux

2.3 Gestionnaire de version et travail collaboratif Git

Bien que n'étant qu'en binôme, travailler en équipe sur un même code sans outil approprié se révèle souvent être un enfer. En effet, la modification d'un fichier peut entrainer des incompatibilités et des bugs dans le reste du programme. Et c'est à ce genre de problèmes que vient répondre un gestionnaire de version.
Tout comme les frameworks graphiques, il existe plusieurs solutions sur le marché, on retiendra essentiellement svn et git. Le choix de git fût immédiat car maitrisé par l'ensemble de l'équipe.

Git a été créé en 2005 par Linus Torvalds, créateur de noyau linux, devant la nécessité de cannaliser et organiser les contributions des milliers de contributeurs de linux. C'est le gestionnaire de version le plus utilisé, et est un réel standard dans le monde du développement logiciel.

FIGURE 2.3 – Logo de github

Git est un outil déscentralisé, un service : il doit être herbergé par un serveur. Là ou il est possible d'être indépendant et d'héberger son propre serveur github, nous avons préferé nous rabattre sur un hébergeur gratuit et public, github.
Ce choix est motivé et par la facilité d'utilisation qu'offre github, mais aussi par la volonté d'avoir un programme libre. Github met ainsi à disposition librement et pour tous notre programme, sous la licence de notre choix [7].

7. Ici MIT

2.4 Générateur de documentation Doxygen

Doxygen est un générateur de documentation à partir du code source. Un des problèmes lié au travail de groupe et auquel git ne répond pas, est la maintanibilité et la lisibilité du code, et c'est là que doxygen apporte des solutions.
Avec une syntaxe claire et rigoureuse de commentaires judicieusement placés sur chaque fonction et instruction utile, le code produit par une personne devient limpide pour une autre personne formée à utiliser doxygen. De plus, la documentation générée (sous format html, pdf..) permet d'avoir une vue d'ensemble sur le code, notamment grace à des graphiques représentant les intéractions entre les classes.

Source 2.1 – Exemple de commentaire utilisant la syntaxe de doxygen

```
/**
 * @brief generate a Node tree representation of a tokenList
 *
 * @param tokens the tokenList to parse
 * @return a Node tree representation of the tokenlist
 */

Node* generateTree(QList<Token> tokens)
```

Le dernier avantage, et pas des moindres, est que pour tout nouveau contributeur dans le projet, l'apprentissage de la structure du logiciel est bien plus rapide que sans documentation. Le choix ayant été fait d'avoir un programme libre, il a semblé naturel de fournir une documentation allant avec le programme.

Un autre choix ayant été fait au niveau des commentaires a été de les rédiger entièrement en Anglais. Dans le monde de l'informatique, l'anglais est omniprésent : au niveau des documentations, des forums, des ressources. Commenter en français implique que la portabilité du programme ne pourra se faire qu'en millieu francophone et qu'aucune contribution ne pourra être apportée par quelqu'un ne maitrisant pas le français. De l'autre côté, l'anglais est compris pratiquement universellement.

Chapitre 3

Algorithmes utilisés

La recherche des algorithmes et leur développement fut l'un des gros morceaux du projet. Cette partie pris une semaine de développement.

3.1 Design pattern : Lexer/Parser

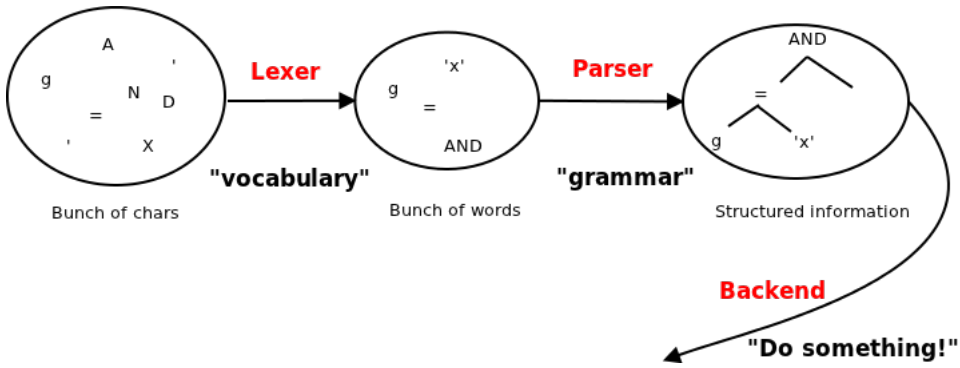

FIGURE 3.1 – Représentation du Lexer/Parser

Le choix a été fait de remplir un des objectifs secondaires du cahier des charges, "l'imbrication libre des instruction".
Le problème a été traité de manière traditionnelle, une utilisant le design pattern lexer/Parser.
Le lexer, est "l'analyseur lexical". Il transorme une série de caractères en "tokens", qui sont des chaines de caractères définies au préalable.
Ainsi, un mot est un token "T_STRING", un chiffre est un token "T_SCALAR" ou encore une matrice un "T_MATRIX". Au niveau du code, ces tokens sont représentés par des objets (voir A.8) ayant chacun la valeur de la chaine de caractère reconnue associée à leur type. Exemple : le chiffre 3 est un objet Token de type "T_SCALAR" et de valeur "3".
La reconnaissance de ces tokens se fait par regex successive sur la chaine de caractère ce que l'une d'entre elle fonctionne.s

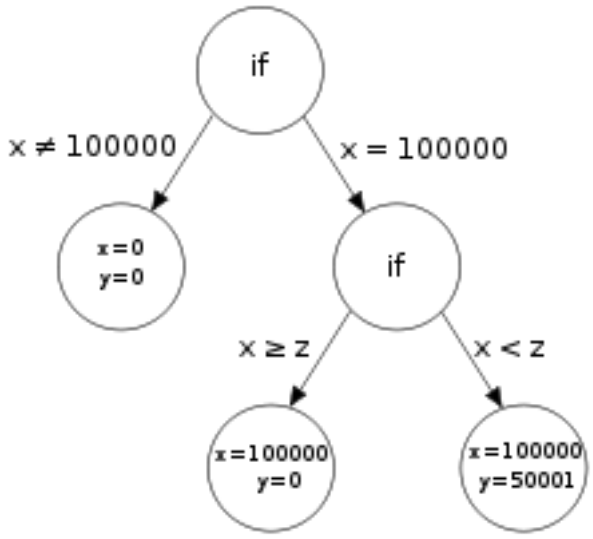

FIGURE 3.2 – Création d'un arbre d'execution. Des noeuds conditionnels sont représentés

Une fois l'expression analysée, il faut l'executer. Le parser passe en revue tous les tokens en essayant de reconnaître des "motifs" d'expression cohérentes. Il les assemble ensuite en Node (voir A.17), qui sont comme leur nom l'indique des noeud d'expression. Un exemple de noeud est le noeud d'assignation, ayant comme motif un nom de variable et une expression séparée par un signe égal.

Le code est pensé de manière modulaire, il est ainsi possible de rajouter d'autres types de noeud comme des noeuds conditionnels par exemple, et ainsi étendre l'analyseur d'expression en un véritable interpréteur de language.
Les Nodes créées, elles sont ensuite assemblées en arbre et ensuite executée. Nous allons maintenant voir par quelle méthode les expressions sont interprétées.

3.2 Execution des expression : Notation Polonaise Inversée

L'execution des expressions peut se faire de plusieurs manières. La première qui a été abordée puis abandonnée par la suite, est la création d'un arbre syntaxique similaire à l'arbre d'éxecution introduit à la section précédente. Le problème avec cette manière de fonctionner est qu'elle est bien plus lourde niveau code et donc source de bugs. Une implémentation de cette méthode a néammoins été produite et reste visible dans l'historique des commits du projet.
L'autre manière d'aborder le problème et qui a été retenue pour le projet est la conversion de l'expression sous sa forme infixe (forme usuelle, utilisant les parenthèses) en notation polonaise inversée, une forme d'expression bien plus simple a executer qui ne comprend ni parenthèses ni priorité d'opérateurs.
Voici un exemple simple d'expression.

$$(2 + 4) * 2 \tag{3.1}$$

Une fois convertis en notation polonaise inversée, celà donne l'expression suivante.

$$24 + 2* \tag{3.2}$$

Le principe est d'empiler opérandes et opérateurs, chaque opérateurs opérant sur les deux opérandes qui le précède.

Cette méthode, très connue et largement utilisée par les compilateurs nous a permis de trouver une implémentation de l'algorithme permettant la conversion de l'expression, et quant à l'éxecution de l'expression, c'est un jeux d'enfant.

Chapitre 4

Fonctionnement du programme

4.1 Interface graphique

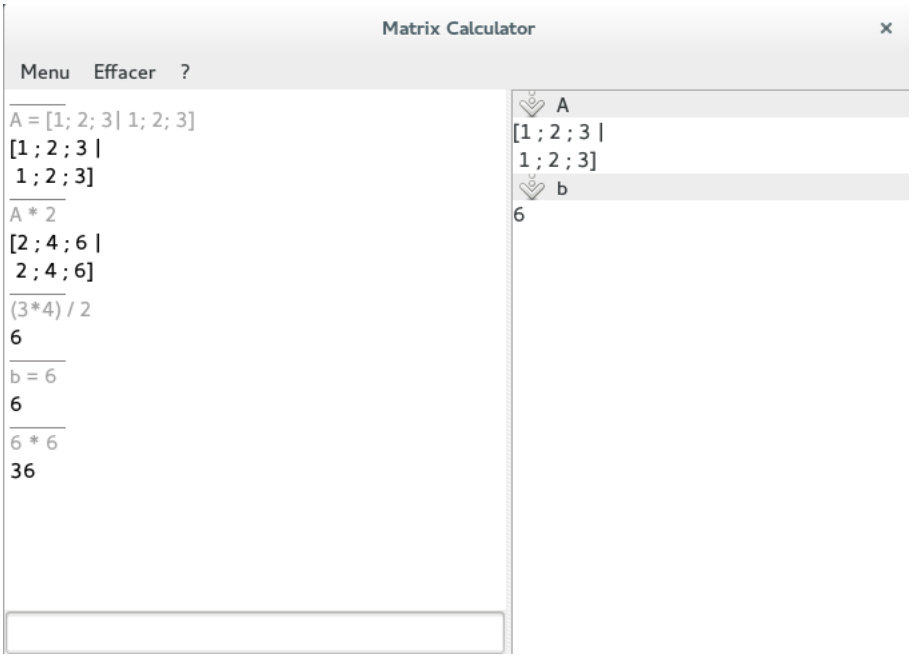

FIGURE 4.1 – Interface du programme

L'interface est composée d'un champ d'entrée de sélection, d'une zone d'affichage comprenant les expressions entrées et leurs resultat (ou erreur) ainsi qu'une gestionnaire de variable. Dans la partie principale, l'expression entrée par l'utilisateur est en gris et les erreurs, s'il y en a, sont affichées en rouges. et le resultat retournée par son expression en noir. Chaque expression est séparée par une ligne noire.
Il est possible à l'aide des flèches de remonter ou redescendre dans l'historique des entrées, en ayant prit soin de sélectionner la zone de texte d'entrée.

4.2 Fonctionnalités

Les variables et leur contenu sont affichés en temps réel sur la droite. L'utilisateur peut vider le contenu des variables en cliquant droit sur l'une d'elle, un menu contextuel apparaît et offre la possibilité de la supprimer.
En cliquant simplement sur le nom de la variable, il est possible d'afficher ou de cacher sont contenu.
La suppression des variables, ainsi que celle de l'historique est disponible dans le menu "effacer".

L'état actuel du programme (contenu des variables ainsi qu'historique des entrées) peut être sauvegardé dans un fichier à l'aide du raccourcis clavier Ctrl+S ou dans le menu "Menu". De la même manière, un état précédent du programme peut être récuperé en ouvrant un fichier avec le raccourcis Ctrl+O ou dans le menu "Menu".

L'assignation à une variable se fait de la manière suivante :

$$nomVariable = [valeur]$$

La valeur peut être un scalaire, la résultante d'une fonction ou une matrice.
Les fonctions implémentées sont :

1. Le déterminant d'une matrice
$$det([matrice])$$
2. L'inverse d'une matrice
$$inv([matrice])$$
3. La résolution d'équation linéaires (scalaires ou matricielles) :
$$solve([A], [B])$$

La syntaxe de définition d'une matrice est la suivante (pour une matrice 2x2)

$$[[valeur]; [valeur]|[valeur]; [valeur]]$$

Le " ;" est le séparateur de colonne et "|" le séparateur de ligne.

Chapitre 5

Conclusion

La réalisation de ce projet a eu à la fois un apport théorique, technique et humain sur les connaissances apportées par NF05. C'est donc une mise en situation réaliste nécessitant la mise en pratique de nos trois grands axes de connaissances universitaires et permettant de renforcer nos acquis.

Dans un premier temps, la délimitation du sujet (prise en compte des contraintes) et les objectifs à atteindre (création du cahier des charges) ainsi que les choix technologiques permettent d'avoir un vrai aperçu du travail d'ingénieur.
La réalisation technique, en plus de renforcer les acquis de cours, a ouvert notre champ de connaissance en nous permettant de découvrir et d'utiliser d'autres technologies.
Enfin, la réalisation d'un projet en groupe met en évidence les différents problèmes liés au travail collaboratif et nous a permis de mettre en oeuvre des solutions efficaces, et ainsi d'obtenir des compétences essentielles pour notre futur métier d'ingénieur et notre travail en entreprise.

En guise de conclusion sur le programme en lui même, nous pouvons dire que la plupart des objectifs qui étaient fixés ont étés remplis. Le cahier des charges est entièrement rempli ; nous regrettons néammoins de ne pas avoir pu implémenter une synthaxe plus "naturelle" quant à l'écriture des matrices. Une idée que nous avions eu était d'avoir une représentation graphique des matrice en tant réelle en utilisant de l'html/css dans un applet webkit. Par manque de temps, nous n'avons pas pu le faire. Le programme est également modulaire : nous voulions rajouter la gestion des nombres complexes, par exemple, et comme pour l'idée précédente, c'est le temps qui nous a fait défaut.

Annexe A

Annexes

A.1 Code

Fichiers source

Source A.1 – mainwindow.cpp

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

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->lineEdit, SIGNAL(returnPressed()), this, SLOT(eval()));
    connect(ui->detailedList, SIGNAL(elementDeleted(QString)), this,
        SLOT(deleteVar(QString)));
    connect(ui->actionClearVars, SIGNAL(triggered()), ui->detailedList,
        SLOT(deleteAll()));
    connect(ui->actionClearAll, SIGNAL(triggered()), ui->detailedList,
        SLOT(deleteAll()));
    connect(ui->actionClearHist, SIGNAL(triggered()), this, SLOT(clearHist()));
    connect(ui->actionClearAll, SIGNAL(triggered()), this, SLOT(clearHist()));
    connect(ui->actionAbout, SIGNAL(triggered()), this, SLOT(about()));
    connect(ui->actionSave, SIGNAL(triggered()), this, SLOT(save()));
    connect(ui->actionOpen, SIGNAL(triggered()), this, SLOT(open()));

    currentPos = 0;
}

MainWindow::~MainWindow()
{
    delete ui;
}


bool MainWindow::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress)
    {
        QKeyEvent *ke = static_cast<QKeyEvent *>(event);

        if (ke->key() == Qt::Key_Up)
        {
            if (this->currentPos > 0)
            {
                this->currentPos--;
                this->ui->lineEdit->setText(this->history[this->currentPos]);
```

```
            }
        }
        else if (ke->key() == Qt::Key_Down)
        {
            if (this->currentPos < this->history.length()-1)
            {
                this->currentPos++;
                this->ui->lineEdit->setText(this->history[this->currentPos]);
            }
        }
    }

    return QWidget::event(event);
}


void MainWindow::memorySync()
{
    QMultiMap<QString, QListWidgetItem*>* list = ui->detailedList->getList();

    foreach (VarNode* currentVar, *(VarNode::getRegistry()))
    {
        QString varName = currentVar->getName();

        if (currentVar->getValue() != NULL) {
            QString varValue = currentVar->getValue()->getValue();

            bool found = false;

            foreach (QListWidgetItem* currentItem, list->values(varName))
            {
                QLabel* currentLabel = (QLabel*)
                    ui->detailedList->itemWidget(currentItem);

                if (currentLabel->text() != varValue)
                {
                    ui->detailedList->UpdateElement(varName, varValue);
                }

                found = true;
                break;
            }

            if (!found)
            {
                ui->detailedList->addElement(varName, varValue);
            }
        }
    }
}


void MainWindow::deleteVar(QString varName)
{
    for (int i = 0; i < VarNode::getRegistry()->length(); i++)
    {
```

```
        VarNode* currentNode = VarNode::getRegistry()->at(i);

        if (currentNode->getName() == varName)
        {
            VarNode::getRegistry()->removeAt(i);
            return;
        }
    }
}

void MainWindow::eval()
{
    QString expression = ui->lineEdit->text();
    if(this->history.isEmpty() || this->history.last() != expression)
        this->history.append(expression);
    this->currentPos = this->history.length();
    Calculable *value;
    Parser parser(expression);

    ui->textBrowser->append("<span
        style=\"color:#858282;text-decoration:underline;font-size:5px;\">     
    ui->textBrowser->append("<span style=\"color:#A3A3A3;\">" + expression +
        "</span>");

    try
    {
        value = parser.run();
        memorySync();
        if (value != NULL)
        {
            ui->textBrowser->append("<span style=\"color:black;\">" +
                (value->getValue().replace('\n', "<br/>").replace(' ', " ")) +
                "</span>");
            ui->lineEdit->setText("");
        }
        else
        {
            throw std::runtime_error("Unknown error");
        }
    }
    catch (std::runtime_error &e)
    {
        ui->textBrowser->append("<span style=\"color:red\">[Error] " +
            QString::fromStdString(e.what()) + "</span>");
    }
    catch (...) {
        ui->textBrowser->append("<div style=\"color:red\">[Error] Caught an unknown
            exception</div>");
    }
    //Scroll to the bottom of the ouput
    ui->textBrowser->moveCursor(QTextCursor::End);
}

void MainWindow::clearHist()
{
    ui->textBrowser->clear();
```

```
    this->history.clear();
    currentPos = 0;
}

void MainWindow::about()
{
    About* ab = new About();
    ab->show();
}

void MainWindow::save()
{
    QString filename = QFileDialog::getSaveFileName(this, "Save session file",
        "session.maf", "Matrix calculator file (*.maf);;All files (*)");
    if(filename.isEmpty())
        return;

    QFile f(filename);
    if(!f.open(QIODevice::WriteOnly | QIODevice::Text))
        return;
    // stores vars, one per line
    foreach(VarNode* var, *(VarNode::getRegistry()))
    {
        if(var != NULL && var->getValue() != NULL)
        {
            QString varName = var->getName();
            QString varStr = var->getValue()->getValue();

            //Remove \n and/or \r that are used to make variables text more
                human-readable
            varStr = varStr.replace('\n',"").replace('\r', "");
            varName = varName.replace('\n',"").replace('\r', "");

            //Write variable to the file
            if(!varStr.toUtf8().isEmpty() && !varName.toUtf8().isEmpty())
            {
                f.write(varName.toUtf8());
                f.write("=");
                f.write(varStr.toUtf8());
                f.write("\n");
            }
        }
    }

    //Jump another line to signal end of variable list
    f.write("\n");

    //History list
    foreach(QString str, this->history)
    {

        //Remove \n and/or \r even if they're not supposed to be there
        str = str.replace('\n',"").replace('\r', "");

        //Write variable to the file
        if(!str.toUtf8().isEmpty())
```

```
        {
            f.write(str.toUtf8());
            f.write("\n");
        }
    }

    //Jump another line to signal end of history list
    f.write("\n");

    //Calculs output
    f.write(QByteArray(this->ui->textBrowser->toHtml().toUtf8()));

    f.close();
}


void MainWindow::open()
{
    QString filename = QFileDialog::getOpenFileName(this, "Open session file",
        "session.maf", "Matrix calculator files (*.maf);;All files (*)");
    if(filename.isEmpty())
        return;

    QFile f(filename);
    if(!f.open(QIODevice::ReadOnly | QIODevice::Text))
        return;

    //Reset current vars and hist
    clearHist();
    ui->detailedList->deleteAll();

    int state = 0;
    QString html;
    //Reload all data
    while (!f.atEnd()) {
        QByteArray line = f.readLine();
        //Remove all endline chars
        line = line.replace('\n',"").replace('\r', "");

        if(line.isEmpty() && state < 2)
        {
            state++;
        }
        else
        {
            switch(state)
            {
                case 0: //Reading vars
                {
                    Parser parser(line);
                    try
                    {
                        parser.run();
                        memorySync();
                    }
                    catch (...) {
```

```
                    ui->textBrowser->append("<span style=\"color:red\">[Error]
                        Could not restore one of the variables from the file :
                        <br/>" + line + "</span>");
                    return;
                }
            }
            break;

            case 1: //Reading hist
            {
                this->history.append(line);
                break;
            }
            case 2: //Reading hist
            {
                html.append(line);
                break;
            }
        }
    }

    }
    this->currentPos = this->history.length();

    //Write html content to the output
    ui->textBrowser->setHtml(html);

    //Scroll to the bottom of the ouput
    ui->textBrowser->moveCursor(QTextCursor::End);
}
```

Source A.2 – detailedlist.cpp

```
#include "detailedlist.hpp"

DetailedList::DetailedList(QWidget *parent) :
    QListWidget(parent)
{
    setSelectionMode(QAbstractItemView::NoSelection);
    this->setContextMenuPolicy(Qt::CustomContextMenu);

    connect(this,SIGNAL(itemClicked(QListWidgetItem*)),this,SLOT(itemClicked(QListWidgetItem*)));
    connect(this,SIGNAL(customContextMenuRequested(const
        QPoint&)),this,SLOT(customContextMenuRequested(const QPoint&)));

}


QMultiMap<QString, QListWidgetItem*>* DetailedList::getList()
{
    return &(this->list);
}


void DetailedList::addElement(QString title, QString content, bool expanded)
{
    if(!list.contains(title))
    {
        int itemLine;
        //Find alphabetical position
        for(itemLine=0; itemLine<list.keys().count();itemLine++)
        {
            if(list.keys().at(itemLine) > title)
            {
                break;
            }
        }


        //Create first line : the title item
        insertItem(itemLine, title);
        list.insert(title, item(itemLine));
        item(itemLine)->setBackground(QPalette().window());
        item(itemLine)->setSizeHint(QSize(item(itemLine)->sizeHint().width(), 20));


        //Create content item
        insertItem(itemLine+1, "");
        list.insert(title, item(itemLine+1));
        //Add a label that can show multiline rich text
        QLabel* contentLabel = new QLabel(content, this);
        contentLabel->setTextInteractionFlags(Qt::TextSelectableByMouse |
            Qt::TextSelectableByKeyboard);
        setItemWidget(item(itemLine+1), contentLabel);
        item(itemLine+1)->setSizeHint(QSize(item(itemLine+1)->sizeHint().width(),
            contentLabel->height()));

        //set default expanded state
```

```
        expandElement(title,expanded);
    }
}

void DetailedList::deleteElement(QString title)
{
    //Delete all items
    foreach(QListWidgetItem* item, list.values(title))
    {
        delete(this->itemWidget(item));
        delete(item);
    }
    //Remove element from the data list
    list.remove(title);
}

void DetailedList::deleteAll()
{
    //Delete all items
    foreach(QString item, list.keys())
    {
        emit elementDeleted(item);
        deleteElement(item);
    }
}

void DetailedList::UpdateElement(QString title, QString content)
{
    //Check if we can find an item associated with the title given
    QListWidgetItem* contentItem = list.values(title).first();
    if(list.contains(title) && itemWidget(contentItem)->metaObject()->className()
        == QString("QLabel"))
    {
        QLabel* contentLabel = (QLabel*)itemWidget(contentItem);
        contentLabel->setText(content);
        contentLabel->adjustSize();
        contentItem->setSizeHint(QSize(contentItem->sizeHint().width(),
            contentLabel->height()));
    }
}

void DetailedList::expandElement(QString title, bool expand)
{
    //Check if we can find an item associated with the title given
    if(list.contains(title))
    {
        QListWidgetItem* titleItem = list.values(title).last();
        QListWidgetItem* contentItem = list.values(title).first();
        if(expand)
        {
            contentItem->setHidden(false);
            titleItem->setIcon(style()->standardIcon(QStyle::SP_ArrowDown));
        }
        else
        {
            contentItem->setHidden(true);
```

```
            titleItem->setIcon(style()->standardIcon(QStyle::SP_ArrowRight));
        }
    }
}


void DetailedList::itemClicked(QListWidgetItem* clicked)
{
    //Check if we can find an item associated with the title given
    if(clicked->text() != "" && list.contains(clicked->text()))
    {
        //Expand if reduced
        QListWidgetItem* contentItem = list.values(clicked->text()).first();
        expandElement(clicked->text(), contentItem->isHidden());
    }
}

void DetailedList::customContextMenuRequested(const QPoint& pos)
{
    QListWidgetItem* clicked = itemAt(pos);
    //Check if we can find an item associated with the title given
    if(clicked->text() != "" && list.contains(clicked->text()))
    {
        QMenu *menu = new QMenu();
        QAction *delAction = new QAction(tr("Delete"), menu);
        delAction->setIcon(style()->standardIcon(QStyle::SP_TrashIcon));
        connect(delAction, SIGNAL(triggered()),this,SLOT(deleteTriggered()));
        menu->addAction(delAction);
        menu->popup(viewport()->mapToGlobal(pos));
        contextMenuOrigin = clicked;
    }
}


void DetailedList::deleteTriggered()
{

    //Emit signal to inform outside of the object that an element will be deleted
    emit elementDeleted(contextMenuOrigin->text());

    //Delete the element
    deleteElement(contextMenuOrigin->text());
}
```

Source A.3 – mainwindow.hpp

```
#ifndef MAINWINDOW_HPP
#define MAINWINDOW_HPP

#include <QMainWindow>
#include <QList>
#include <QEvent>
#include <QKeyEvent>
#include <QString>
#include <stdexcept>
#include <QFileDialog>

#include "about.h"
#include "lib/parser.h"
#include "lib/calculable.h"
#include "lib/varNode.h"


namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
        Q_OBJECT

    public:
        explicit MainWindow(QWidget *parent = 0);
        ~MainWindow();
        virtual bool event(QEvent *event);

    private:
        void memorySync();
        int currentPos;
        QList<QString> history;
        Ui::MainWindow *ui;

    private slots:
        void deleteVar(QString varName);
        void eval();
        void clearHist();
        void about();
        void save();
        void open();
};

#endif // MAINWINDOW_HPP
```

Source A.4 – about.cpp

```
#include "about.h"
#include "ui_about.h"

About::About(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::About)
{
    ui->setupUi(this);
}

About::~About()
{
    delete ui;
}
```

Source A.5 – detailedlist.hpp

```
#ifndef DETAILEDLIST_HPP
#define DETAILEDLIST_HPP

#include <QListWidget>
#include <QLabel>
#include <QDateTime>
#include <QMenu>
#include <QDebug>

class DetailedList : public QListWidget
{
    /**
    * @class DetailedList
    * @brief A list widget with details when you click on the title line
    *
    * A list widget with details when you click on the title line with a
    * triangle or an arrow that indicate current state.
    * Each element is composed by to QListWidgetItem : title and content.
    * The list is sorted by title alphabetical order
    *
    * **Warning** : As each element is indexed by the title, it has to
    * be unique.
    *
    */

    Q_OBJECT
    public:

        /**
        * @brief Constructor
        *
        * Construct an empty DetailedList with the given *parent*
        *
        * @param parent
        */
        explicit DetailedList(QWidget *parent = 0);

        /**
        * @brief Add a new element to the list
        *
        * Add a new element to the list
        *
        * @param title   Plain text title that is allways visible
        * @param content  Rich text content that will be visible when the user click
            on the line
        * @param expanded Choose if the new element will be expanded or not
        */
        void addElement(QString title, QString content, bool expanded = true);

        /**
        * @brief Delete an element from the list
        *
        * Delete an element from the list
        *
        * @param title  Plain text title of the element that you want to delete
```

```
    */
    void deleteElement(QString title);


    /**
    * @brief Update an element from the list
    *
    * Update an element from the list
    *
    * @param title  Plain text title of the element that you want to update
    * @param content  The new content of the element
    */
    void UpdateElement(QString title, QString content);


    /**
    * @brief Expand or reduce an element of the list
    *
    * Expand or reduce an element of the list
    *
    * @param title  Plain text title of the element that you want to update
    * @param expand Choose if the element will be expanded or not
    */
    void expandElement(QString title, bool expand = true);


    QMultiMap<QString, QListWidgetItem*>* getList();

public slots:

    /**
     * @brief Delete all elements from the list
     */
    void deleteAll();
protected slots:
    /**
    * @brief Slot triggered when a click is made on an item
    *
    * Slot triggered when a click is made on an item. This is used to expand and
        reduce elements when you click on a title.
    *
    * @param clicked A pointer to the clicked item
    */
    void itemClicked(QListWidgetItem* clicked);

    /**
    * @brief Slot triggered when a right click is made
    *
    * Slot triggered when a right click is made. This will add a context menu on
        elements that will allow user to delete the element.
    *
    * @param pos  Position of the mouse when the user right click
    */
    void customContextMenuRequested(const QPoint& pos);

    /**
```

```
        * @brief Slot triggered when a click is made on the delete button of the
            context menu
        *
        * Slot triggered when a click is made on the delete button of the context
            menu. This will emit the but public elementDeleted() signal and then
            remove the element from the list.
        */
        void deleteTriggered();

    private:
        /**
        * @brief Help to know where come from a context menu
        *
        * A pointer to a QListWidgetItem that help to know where come from a context
            menu
        */
        QListWidgetItem* contextMenuOrigin;
        /**
        * @brief A list that associate title with items
        *
        * A list that associate title with items (the title item and the content
            item)
        */
        QMultiMap<QString, QListWidgetItem*> list;

    signals:
        /**
        * @brief Signal emitted when the user delete an element
        *
        * Signal emitted when the user delete an element
        *
        * @param title The title of the deleted element
        */
        void elementDeleted(QString title);

};

#endif // DETAILEDLIST_HPP
```

Source A.6 – about.h

```
#ifndef ABOUT_H
#define ABOUT_H

#include <QDialog>

namespace Ui {
class About;
}

class About : public QDialog
{
    Q_OBJECT

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

private:
    Ui::About *ui;
};

#endif // ABOUT_H
```

Source A.7 – main.cpp

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

/**
 * @mainpage Project documentation
 *
 * \section intro_sec Introduction
 *
 * This is the documentation of our Matrix Calculator.
 *
 * This project has been made by Axel Mousset and Aurelien Labate
 * when they were following the NF05 course (Introduction to C)
 * at the University of Technology of Troyes.
 *
*/

int main(int argc, char *argv[])
{

    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

Source A.8 – lib/token.h

```
#ifndef TOKEN_H
#define TOKEN_H

#include <QDebug>
#include <QString>


enum TokenKind
{
    T_NULL,
    T_ASSIGNMENT,
    T_WHITESPACE,
    T_SOLVE,
    T_MULTIPLY,
    T_DIVIDE,
    T_SUM,
    T_SUB,
    T_MODULO,
    T_POW,
    T_SCALAR,
    T_MATRIX,
    T_PARENTHESIS_LEFT,
    T_PARENTHESIS_RIGHT,
    T_STRING,
    T_COMMA
};


class Token
{
    public:

        /**
         * @brief constructor
         *
         * @param kind kind of token
         * @param value value of token
         */
        Token(TokenKind kind, QString value="");

        /**
         * @brief destructor
         */
        ~Token();

        /**
         * @brief kind accessor
         * @return kind of token
         */
        TokenKind getKind() const;

        /**
         * @brief value accessor
         * @return value of token
         */
```

```
        QString getValue() const;

        /**
         * @brief setValue value setter
         * @param value value to set
         */
        void setValue(QString value);

        /**
         * @brief setKind kind setter
         * @param kind kind to set
         */
        void setKind(TokenKind kind);


    protected:
        TokenKind kind;
        QString value;
};

#endif // TOKEN_H
```

Source A.9 – lib/calculables/matrix.cpp

```
#include "matrix.h"



Matrix::Matrix(QString value)
{
    N=-1;M=-1;
    this->setValue(value);
}

Matrix::Matrix(Matrix &value)
{
    N=-1;M=-1;
    this->setValue(&value);
}

Matrix::Matrix()
{
    N=-1;M=-1;
    this->setM(1);
    this->setN(1);
}

QString Matrix::getValue()
{

    QString out = "[";

    for(int i=0; i < this->getM(); i++)
    {
        if(i != 0)
            out += " |\n ";

        for(int j=0; j < this->getN(); j++)
        {
            if(j != 0)
                out += " ; ";

            out += QString::number(this->getCell(i, j));
        }
    }


    return (out + ']');
}

QVector< QVector<double> > Matrix::getRawValue()
{
    return this->value;
}

void Matrix::setRawValue(QVector< QVector<double> > newValue)
{
    this->value = newValue;
    //Update matrix size
```

```
    this->setM(this->getM());
    this->setN(this->getN());
}

void Matrix::setValue(QString newValue)
{

    //Remove []
    newValue.remove('[');
    newValue.remove(']');

    //Parse matrix format a;b!c;d
    QStringList rows = newValue.split(QRegExp("[!\\|]"));
    this->setM(rows.size());

    //iterate over rows
    for(int i=0; i < this->getM(); i++)
    {
        QStringList cols = rows[i].split(';');

        //Resize matrix if first line
        if(i == 0)
            this->setN(cols.size());
        else if(cols.size() != this->N)
            throw std::runtime_error("Matrix [" + newValue.toStdString() + "] cannot
                be parsed : Row " + QString::number(i+1).toStdString() + " doens't
                have the same column number as the precedent row");

        //iterate over columns
        for(int j=0; j<this->getN() ; j++)
        {
            //Parse cell
            //TODO GET VARS VALUES
            Parser parser(cols[j]);
            Calculable &colVal = *(parser.run());

            //Errors
            if(colVal.getType() != T_SCALAR)
            {
                throw std::runtime_error("Only scalar types can be inside matrix not
                    " + colVal.getTypeStr() + " (=" + colVal.getValue().toStdString()
                    + ")");

            }
            else if(colVal.getValue() == NULL)
                throw std::runtime_error("Unknown error happend during matrix
                    computation");

            //insert value
            this->setCell(i, j, dynamic_cast<Scalar&>(colVal).getRawValue());

        }
    }
}

void Matrix::setValue(Matrix *A)
```

```
{
    this->setM(A->getM());
    this->setN(A->getN());
    this->setRawValue(A->getRawValue());
}

Calculable* Matrix::operator*(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        Matrix& a2 = dynamic_cast<Matrix&>(a);
        Matrix *out = new Matrix();

        //Set new matrix size
        if(this->getM() == a2.getN())
        {
            out->setM(a2.getM());
            out->setN(this->getN());
        }
        else
        {
            throw std::runtime_error("Cannot multiply a matrix "
                                     + QString::number(a2.getM()).toStdString() + "x" +
                                        QString::number(a2.getN()).toStdString()
                                     + " with a matrix "
                                     + QString::number(this->getM()).toStdString() +
                                        "x" +
                                        QString::number(this->getN()).toStdString());
        }

        //set values
        for(int i=0; i < out->getM(); i++)
        {
            for(int j=0; j < out->getN() ; j++)
            {
                double sum = 0;
                for(int k=0; k < this->getM(); k++)
                {
                    sum += a2.getCell(i, k) * this->getCell(k, j);
                }
                out->setCell(i, j, sum);
            }
        }
        return out;
    }
    else if(a.getType() == T_SCALAR)
    {
        Scalar& a2 = dynamic_cast<Scalar&>(a);
        Matrix *out = new Matrix();

        //Set new matrix size
        out->setM(this->getM());
        out->setN(this->getN());

        //set values
        for(int i=0; i < out->getM(); i++)
```

```
        {
            for(int j=0; j < out->getN() ; j++)
            {
                out->setCell(i, j, this->getCell(i, j) * a2.getRawValue());
            }
        }
        return out;
    }
    else
    {
        throw std::runtime_error("Cannot use the operator * between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
    }
    return NULL;
}

Calculable* Matrix::operator+(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        Matrix& a2 = dynamic_cast<Matrix&>(a);
        Matrix *out = new Matrix();

        //Set new matrix size
        if(this->getM() == a2.getM() && this->getN() == a2.getN())
        {
            out->setM(this->getM());
            out->setN(this->getN());
        }
        else
        {
            throw std::runtime_error("Cannot sum a matrix "
                                     + QString::number(a2.getM()).toStdString() + "x" +
                                       QString::number(a2.getN()).toStdString()
                                     + " with a matrix "
                                     + QString::number(this->getM()).toStdString() +
                                       "x" +
                                       QString::number(this->getN()).toStdString());
        }

        //set values
        for(int i=0; i < out->getM(); i++)
        {
            for(int j=0; j < out->getN() ; j++)
            {
                out->setCell(i, j, this->getCell(i, j) + a2.getCell(i, j));
            }
        }
        return out;
    }
    else
    {
        throw std::runtime_error("Cannot use the operator + between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
    }
    return NULL;
```

```
}

Calculable* Matrix::operator-(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        Matrix& a2 = dynamic_cast<Matrix&>(a);
        Matrix *out = new Matrix();

        //Set new matrix size
        if(this->getM() == a2.getM() && this->getN() == a2.getN())
        {
            out->setM(this->getM());
            out->setN(this->getN());
        }
        else
        {
            throw std::runtime_error("Cannot sum a matrix "
                                     + QString::number(a2.getM()).toStdString() + "x" +
                                       QString::number(a2.getN()).toStdString()
                                     + " with a matrix "
                                     + QString::number(this->getM()).toStdString() +
                                       "x" +
                                       QString::number(this->getN()).toStdString());
        }

        //set values
        for(int i=0; i < out->getM(); i++)
        {
            for(int j=0; j < out->getN() ; j++)
            {
                out->setCell(i, j, a2.getCell(i, j) - this->getCell(i, j));
            }
        }
        return out;
    }
    else
    {
        throw std::runtime_error("Cannot use the operator - between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
    }
    return NULL;
}

int Matrix::getM() {
    return this->M;
}
int Matrix::getN() {
    return this->N;
}
void Matrix::setM(int M) {

    if(M >= 1)
    {
        this->M = M;
        value.resize(M);
```

```
        if(N > 0)
        {
            for(int i = 0; i<M; i++)
            {
                value[i].resize(N);
            }
        }
    }
}
void Matrix::setN(int N) {
    if(N >= 1)
    {
        this->N = N;
        if(this->M > 0)
        {
            for(int i = 0; i<M; i++)
                value[i].resize(N);
        }
    }
}
double Matrix::getCell(const int i, const int j) {
    if(0 <= i && i < this->M && 0 <= j && j < this->N)
        return this->value[i][j];
    return 0;
}
void Matrix::setCell(const int i, const int j, const double value)
{
    if(0 <= i && i < this->M && 0 <= j && j < this->N)
        this->value[i][j] = value;
}

std::string Matrix::getTypeStr()
{
    return "matrix";
}

TokenKind Matrix::getType()
{
    return T_MATRIX;
}
```

Source A.10 – lib/calculables/scalar.h

```
#ifndef SCALAR_H
#define SCALAR_H

#include "../calculable.h"
#include "matrix.h"
#include <cmath>

class Scalar : public Calculable
{
public:

    /**
     * @brief constructor
     *
     * @param value - the string value
     */
    Scalar(QString value);

    /**
     * @brief constructor
     *
     * @param value the raw value
     */
    Scalar(double value);

    /**
     * @brief Copy constructor
     *
     * @param value the raw value
     */
    Scalar(Scalar &value);

    /**
     * @brief value accessor
     *
     * @return the value of the Calculable
     */
    QString getValue();

    /**
     * @brief value accessor
     *
     * @return the raw value of the Calculable
     */
    double getRawValue();

    /**
     * @brief value setter
     *
     * @param newValue the value to set
     */
    void setValue(QString newValue);
    /**
     * @brief value setter
     *
```

```
 * @param newValue the value to set
 */
void setRawValue(double newValue);

/**
 * @brief overload operator * between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator*(Calculable &a);

/**
 * @brief overload operator / between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator/(Calculable &a);

/**
 * @brief overload operator + between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator+(Calculable &a);

/**
 * @brief overload operator - between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator-(Calculable &a);

/**
 * @brief overload operator % between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator%(Calculable &a);

/**
 * @brief overload operator ^ between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator^(Calculable &a);


/**
 * @brief Define the type of the element as a string
 */
```

```
    std::string getTypeStr();

    /**
     * @brief Define the type of the element as a TokenKind from token.h
     */
    TokenKind getType();

protected:
    double value;
};

#endif // SCALAR_H
```

Source A.11 – lib/calculables/matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include <QList>
#include <QStringList>
#include <stdexcept>
#include <QGenericMatrix>
#include "../calculable.h"
#include "../parser.h"
#include "../token.h"

class Matrix : public Calculable
{

public:

    /**
     * @brief constructor
     *
     * @param value - the string value
     */
    Matrix(QString value);

    /**
     * @brief Copy constructor
     *
     * @param value - the matrix
     */
    Matrix(Matrix &value);

    /**
     * @brief Construct a matrix of one per one with the value 0
     */
    Matrix();

    /**
     * @brief value accessor
     *
     * @return the value of the Calculable
     */
    QString getValue();

    /**
     * @brief Raw value accessor
     *
     * @return the raw value of the Calculable
     */
    QVector< QVector<double> > getRawValue();

    /**
     * @brief value setter
     *
     * @param newValue the value to set
     */
    void setValue(QString newValue);
```

```
/**
 * @brief raw value setter
 *
 * @param newValue the value to set
 */
void setRawValue(QVector< QVector<double> > newValue);

/**
 * @brief Copy setter
 *
 * @param newValue the value to set
 */
void setValue(Matrix *newValue);

/**
 * @brief overload operator * between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator*(Calculable &a);

/**
 * @brief overload operator - between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator+(Calculable &a);

/**
 * @brief overload operator + between two Calculable
 *
 * @param a a Calculable
 * @return a Calculable
 */
Calculable* operator-(Calculable &a);

/**
 * @brief Get the number of rows of the matrix
 *
 * @return The number of rows
 */
int getM();
/**
 * @brief Get the number of columns of the matrix
 *
 * @return The number of columns
 */
int getN();
/**
 * @brief Set the number of rows of the matrix
 *
 * @param The new number of rows.
 */
```

```
    void setM(int M);
    /**
     * @brief Set the number of columns of the matrix
     *
     * @param The new number of columns.
     */
    void setN(int N);

    /**
     * @brief Get the raw value of cellule
     *
     * @param i - The row of the cell between 0 and M-1
     * @param j - The row of the cell between 0 and N-1
     * @return The raw value of the cellule
     */
    double getCell(const int i, const int j);

    /**
     * @brief Set the raw value of cellule
     *
     * @param i - The row of the cell between 0 and M-1
     * @param j - The row of the cell between 0 and N-1
     * @param value - The new raw value of the cellule
     */
    void setCell(const int i, const int j, const double value);

    /**
     * @brief Define the type of the element as a string
     */
     std::string getTypeStr();

    /**
     * @brief Define the type of the element as a TokenKind from token.h
     */
     TokenKind getType();

protected:
    /**
     * @brief Handle the matrix raw value
     */
    QVector< QVector<double> > value;

    /**
     * @brief Give the number of rows of the matrix
     */
    int M;

    /**
     * @brief Give the number of columns of the matrix
     */
    int N;
};

#endif // MATRIX_H
```

Source A.12 – lib/calculables/scalar.cpp

```
#include "scalar.h"


Scalar::Scalar(QString value)
{
    this->setValue(value);
}
Scalar::Scalar(double value)
{
    this->setRawValue(value);
}
Scalar::Scalar(Scalar& value)
{
    this->setRawValue(value.getRawValue());
}

QString Scalar::getValue()
{
    return QString::number(this->value);
}
double Scalar::getRawValue()
{
    return this->value;
}

void Scalar::setValue(QString newValue)
{
    bool ok;
    this->value = newValue.toDouble(&ok);
    if(!ok)
    {
        throw std::runtime_error("Could not convert " + newValue.toStdString() + "
            to scalar type");
    }
}

void Scalar::setRawValue(double newValue)
{
    this->value = newValue;
}

Calculable* Scalar::operator*(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        return new Scalar(dynamic_cast<Scalar&>(a).getRawValue() * this->value);
    }
    else if(a.getType() == T_MATRIX)
    {
        return a.operator*(*this);
    }
    else
    {
        throw std::runtime_error("Cannot use the operator * between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
```

```
        return NULL;
    }
}

Calculable* Scalar::operator/(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        return new Scalar(dynamic_cast<Scalar&>(a).getRawValue() / this->value);
    }
    else if(a.getType() == T_MATRIX)
    {

        Matrix& a2 = dynamic_cast<Matrix&>(a);
        Matrix *out = new Matrix();

        //Set new matrix size
        out->setM(a2.getM());
        out->setN(a2.getN());

        //set values
        for(int i=0; i < out->getM(); i++)
        {
            for(int j=0; j < out->getN() ; j++)
            {
                out->setCell(i, j, a2.getCell(i, j) / this->getRawValue());
            }
        }
        return out;
    }
    else
    {
        throw std::runtime_error("Cannot use the operator / between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
        return NULL;
    }
}

Calculable* Scalar::operator-(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        return new Scalar(dynamic_cast<Scalar&>(a).getRawValue() - this->value);
    }
    else
    {
        throw std::runtime_error("Cannot use the operator - between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
        return NULL;
    }
}
Calculable* Scalar::operator+(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        return new Scalar(dynamic_cast<Scalar&>(a).getRawValue() + this->value);
```

```
    }
    else
    {
        throw std::runtime_error("Cannot use the operator + between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
        return NULL;
    }
}

Calculable* Scalar::operator%(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        return new Scalar(fmod(dynamic_cast<Scalar&>(a).getRawValue(), this->value));
    }
    else
    {
        throw std::runtime_error("Cannot use the operator % between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
        return NULL;
    }
}
Calculable* Scalar::operator^(Calculable &a)
{
    if(this->getType() == a.getType())
    {
        return new Scalar(pow(dynamic_cast<Scalar&>(a).getRawValue(), this->value));
    }
    else if(a.getType() == T_MATRIX)
    {

        Matrix& a2 = dynamic_cast<Matrix&>(a);
        if(a2.getM() != a2.getN())
            throw std::runtime_error("Cannot use the operator ^ with a non-square
                matrix.");
        else
        {
            if(std::floor(this->getRawValue()) != this->getRawValue()) //is not
                integer
            {
                throw std::runtime_error("Cannot use the operator ^ between a " +
                    a.getTypeStr() + " and a floating point number. Only integer are
                    supported.");
            }

            if(this->getRawValue() == 0)
            {
                return MatrixLib::identity(a2.getN());
            }
            else if(this->getRawValue() < 0)
            {
                Scalar tmp(*this);
                tmp.setRawValue(tmp.getRawValue()*(-1));
                return tmp^(*MatrixLib::inv(&a2));
            }
```

```
            Matrix out = a2;

            //Set new matrix size
            out.setM(a2.getM());
            out.setN(a2.getN());

            //set values
            for(double i = 1; i < this->getRawValue(); i++)
            {
                out.setValue(dynamic_cast<Matrix*>(out*a2));
            }

            return (new Matrix(out));
        }
    }
    else
    {
        throw std::runtime_error("Cannot use the operator ^ between a " +
            a.getTypeStr() + " and a " + this->getTypeStr());
        return NULL;
    }
}

std::string Scalar::getTypeStr()
{
    return "scalar";
}

TokenKind Scalar::getType()
{
    return T_SCALAR;
}
```

Source A.13 – lib/assignationNode.cpp

```
#include "assignationNode.h"


AssignationNode::AssignationNode(VarNode *variable, ExpressionNode *expression)
{
    this->variable = variable;
    this->expression = expression;
}


AssignationNode::~AssignationNode() {}


Calculable* AssignationNode::execute()
{
    Calculable *value = this->expression->execute();
    this->variable->setValue(value);

    return value;
}


VarNode* AssignationNode::getVariable() const
{
    return this->variable;
}


ExpressionNode* AssignationNode::getExpression() const
{
    return this->expression;
}

QString AssignationNode::toString() const
{
    return "VarNode name: " + this->getVariable()->toString() + " value: " +
        this->getExpression()->toString();
}
```

Source A.14 – lib/varNode.cpp

```
#include "varNode.h"


VarNode::VarNode(QString varName, Calculable* value)
{
    this->varName = varName;
    this->value = value;
}


VarNode::~VarNode() {}


VarNode* VarNode::getVar(QString reference)
{
    foreach (VarNode *node, *(VarNode::getRegistry()))
    {
        if (node->getName() == reference)
        {
            return node;
        }
    }

    VarNode *newNode = new VarNode(reference, NULL);
    VarNode::getRegistry()->append(newNode);
    return newNode;
}


QList<VarNode *>* VarNode::getRegistry()
{
    return VarNode::registry;
}


QList<VarNode *>* VarNode::initializeRegistry()
{
    return new QList<VarNode *>();
}

QList<VarNode *>* VarNode::registry = VarNode::initializeRegistry();


Calculable* VarNode::execute()
{
    if (this->value != NULL)
    {
        return this->value;
    }
    return NULL;
}


QString VarNode::toString() const
{
```

```
    return "VarNode name: " + this->getName() + " value: " +
        this->getValue()->toString();
}


void VarNode::setValue(Calculable *value)
{
    this->value = value;
}


Calculable* VarNode::getValue() const
{
    return this->value;
}


QString VarNode::getName() const
{
    return this->varName;
}
```

Source A.15 – lib/calculable.h

```
#ifndef CALCULABLE_H
#define CALCULABLE_H

#include <QDebug>
#include <stdexcept>
#include <typeinfo>
#include "token.h"


class Calculable
{
    public:

        /**
         * @brief constructor
         *
         * @param value the calculable value
         */
        Calculable(QString value);

        /**
         * @brief constructor
         */
        Calculable();
        /**
         * @brief destructor
         * @details [long description]
         */
        ~Calculable();

        /**
         * @brief value accessor
         *
         * @return the value of the Calculable
         */
        virtual QString getValue();

        /**
         * @brief value setter
         *
         * @param newValue the value to set
         */
        virtual void setValue(QString newValue);

        /**
         * @brief toString method
         *
         * @return a QString representation of the Node
         */
        virtual QString toString();

        /**
         * @brief overload operator * between two Calculable
         *
         * @param a a Calculable
```

```
         * @return a Calculable
         */
        virtual Calculable* operator*(Calculable &a);

        /**
         * @brief overload operator / between two Calculable
         *
         * @param a a Calculable
         * @return a Calculable
         */
        virtual Calculable* operator/(Calculable &a);

        /**
         * @brief overload operator + between two Calculable
         *
         * @param a a Calculable
         * @return a Calculable
         */
        virtual Calculable* operator+(Calculable &a);

        /**
         * @brief overload operator - between two Calculable
         *
         * @param a a Calculable
         * @return a Calculable
         */
        virtual Calculable* operator-(Calculable &a);

        /**
         * @brief overload operator % between two Calculable
         *
         * @param a a Calculable
         * @return a Calculable
         */
        virtual Calculable* operator%(Calculable &a);

        /**
         * @brief overload operator ^ between two Calculable
         *
         * @param a a Calculable
         * @return a Calculable
         */
        virtual Calculable* operator^(Calculable &a);

        /**
         * @brief Define the type of the element as a string
         */
         virtual std::string getTypeStr() = 0;

        /**
         * @brief Define the type of the element as a TokenKind from token.h
         */
         virtual TokenKind getType() = 0;

};
```

```
#endif // CALCULABLE_H
```

Source A.16 – lib/lexer.cpp

```
#include "lexer.h"


Lexer::Lexer(QString source)
{
    this->source = source;
}


Lexer::~Lexer() {}


QList<Token> Lexer::run()
{
    QList<Token> tokens;
    int offset = 0;

    while (offset <= this->source.length()-1)
    {
        Token result = this->match(this->source, offset);

        if (result.getValue() == "")
        {
            throw std::runtime_error("Unable to parse the character " +
                QString::number(offset+1).toStdString());
            qWarning() << "Unable to parse the element, offset: " << offset;
            emit lexerError(this->source, offset);
            return tokens;
        }

        if (result.getKind() != T_WHITESPACE && result.getKind() != T_NULL)
        {
            tokens.append(result);
        }

        offset += result.getValue().length();
    }

    return tokens;
}


QMap<TokenKind, QRegExp> Lexer::initializeTokens()
{
    QMap<TokenKind, QRegExp> map;

    map.insert(T_ASSIGNMENT, QRegExp("^(=)"));
    map.insert(T_WHITESPACE, QRegExp("^(\\s+)"));
    map.insert(T_SOLVE, QRegExp("^(::)"));
    map.insert(T_MULTIPLY, QRegExp("^(\\*)"));
    map.insert(T_DIVIDE, QRegExp("^(/)"));
    map.insert(T_SUM, QRegExp("^(\\+)"));
    map.insert(T_SUB, QRegExp("^(-)"));
    map.insert(T_MODULO, QRegExp("^(%)"));
    map.insert(T_POW, QRegExp("^(\\^)"));
```

```
    map.insert(T_SCALAR, QRegExp("^([0-9][\\.0-9]*)"));
    map.insert(T_MATRIX, QRegExp("^(\\[[^\\[\\]]+\\])"));
    map.insert(T_PARENTHESIS_LEFT, QRegExp("^(\\()"));
    map.insert(T_PARENTHESIS_RIGHT, QRegExp("^(\\))"));
    map.insert(T_STRING, QRegExp("^([A-Za-z][A-Za-z0-9_]*)"));
    map.insert(T_COMMA, QRegExp("^(,)"));

    return map;
}


QMap<TokenKind, QRegExp> Lexer::tokens = Lexer::initializeTokens();


Token Lexer::match(QString line, int offset)
{
    QString string = line.right(line.length() - offset);

    foreach (TokenKind kind, Lexer::tokens.keys())
    {
        QRegExp value = Lexer::tokens.value(kind);
        int pos = value.indexIn(string);
        if (pos > -1)
        {
            QString match = value.cap(1);
            return Token(kind, match);
        }

    }

    return Token(T_NULL, NULL);
}
```

Source A.17 – lib/node.h

```
#ifndef NODE_H
#define NODE_H

#include <QObject>
#include <QDebug>
#include <QString>
#include <QList>
#include <QMap>


#include <lib/calculable.h>
#include <lib/token.h>


class Node : public QObject
{
    Q_OBJECT

    public:

        /**
         * @brief constructor
         */
        Node();

        /**
         * @brief destructor
         */
        ~Node();

        /**
         * @brief Pure virtual method. Execute the node depending on its type (use
            polymorphism)
         */
        virtual Calculable* execute() = 0;


        /**
         * @brief Pur virtual method. Return a string-based representation of the
            node.
         */
        virtual QString toString() const = 0 ;
};

#endif // NODE_H
```

Source A.18 – lib/calculable.cpp

```
#include "calculable.h"

Calculable::Calculable(QString value)
{
    this->setValue(value);
}

Calculable::Calculable()
{
}



Calculable::~Calculable() {}

QString Calculable::getValue()
{
    throw std::runtime_error("Cannot get value of an element of type " +
        getTypeStr());
}


void Calculable::setValue(QString newValue)
{
    throw std::runtime_error("Cannot set value of an element of type " +
        getTypeStr());
}

QString Calculable::toString()
{
     return QString(getTypeStr().c_str()) + " value: " + this->getValue();
}

Calculable* Calculable::operator*(Calculable &a)
{
    throw std::runtime_error("Cannot use the operator * with an element of type " +
        getTypeStr());
}


Calculable* Calculable::operator/(Calculable &a)
{
    throw std::runtime_error("Cannot use the operator / with an element of type " +
        getTypeStr());
}


Calculable* Calculable::operator-(Calculable &a)
{
    throw std::runtime_error("Cannot use the operator - with an element of type " +
        getTypeStr());
}


Calculable* Calculable::operator+(Calculable &a)
```

```
{
    throw std::runtime_error("Cannot use the operator + with an element of type " +
        getTypeStr());
}


Calculable* Calculable::operator%(Calculable &a)
{
    throw std::runtime_error("Cannot use the operator % with an element of type " +
        getTypeStr());
}


Calculable* Calculable::operator^(Calculable &a)
{
    throw std::runtime_error("Cannot use the operator ^ with an element of type " +
        getTypeStr());
}
```

Source A.19 – lib/parser.cpp

```
#include "parser.h"



Parser::Parser(QString source)
{
    this->source = source;
    this->lexer = new Lexer(source);
}


Parser::~Parser() {}


Calculable* Parser::run()
{
    QList<Token> tokens = this->lexer->run();

    if (tokens.length() == 0)
    {
        return NULL;
    }

    int assignationNumber = 0;

    foreach (Token token, tokens)
    {
        if (token.getKind() == T_ASSIGNMENT)
        {
            assignationNumber++;
        }
    }

    if (assignationNumber > 1)
    {
        throw std::runtime_error("Multiple assignation");
        return NULL;
    }
    else
    {
        Node *tree = this->generateTree(tokens);

        if (tree == NULL)
            return NULL;

        return tree->execute();
    }
}


Node* Parser::generateTree(QList<Token> tokens)
{
    if (tokens.length() == 1)
    {
        Token token = tokens[0];
```

```
        TokenKind kind = token.getKind();


        if (kind == T_SCALAR || kind == T_MATRIX || kind == T_STRING)
        {
            ExpressionNode* node = new ExpressionNode(tokens);
            return node;
        }
        else
        {
            throw std::runtime_error("Operator alone");
            return NULL;
        }
    }
    else
    {
        int assignementPos = 0;

        for (int i = 0; i < tokens.length(); i++)
        {
            //Search for an assignement
            if (!assignementPos && tokens[i].getKind() == T_ASSIGNMENT)
            {
                //found
                assignementPos = i;
                break;
            }
        }

        if (!assignementPos)
        {
            return new ExpressionNode(tokens);
        }
        else
        {
            //Create assignation node. structure : "varName := expression"
            //
            QList<Token> varName = tokens.mid(0, assignementPos);

            if (varName.length() > 1 || varName[0].getKind() != T_STRING ||
                (assignementPos >= tokens.length()-1))
            {
                throw std::runtime_error("Invalid syntax for assignment");
                return NULL;
            }

            ExpressionNode* right = (ExpressionNode*)
                generateTree(tokens.mid(assignementPos+1, tokens.length()-1));
            VarNode *varNode = VarNode::getVar(varName[0].getValue());


            AssignationNode *assignationNode = new AssignationNode(varNode, right);
            return assignationNode;
        }
    }
}
```

```
bool Parser::isFunction(Token token)
{
    QString value = token.getValue().toLower();

    return (value == "cof"
            || value == "det"
            || value == "trans"
            || value == "co"
            || value == "i"
            || value == "inv"
            || value == "trace"
            || value == "norm"
            || value == "solve");
}
```

Source A.20 – lib/token.cpp

```
#include "token.h"


Token::Token(TokenKind kind, QString value)
{
    this->kind = kind;
    this->value = value;
}


Token::~Token() {}


TokenKind Token::getKind() const
{
    return this->kind;
}


QString Token::getValue() const
{
    return this->value;
}


void Token::setValue(QString value)
{
    this->value = value;
}


void Token::setKind(TokenKind kind)
{
    this->kind = kind;
}
```

Source A.21 – lib/operator.h

```
#ifndef OPERATOR_H
#define OPERATOR_H

#include <QMap>
#include "token.h"


enum Associativity
{
    LEFT_ASSOC,
    RIGHT_ASSOC
};


class Operator
{
    public:

        /**
         * @brief constructor
         *
         * @param precedence precedence of the operator
         * @param associativity associativity type of the operator
         */
        Operator(int precedence, Associativity associativity);

        /**
         * @brief destructor
         */
        ~Operator();


    public:

        /**
         * @brief determine if a token is an operator or a function
         *
         * @param token token to test
         * @return nature of the token
         */
        static bool isOperator(Token token);

        static QMap<int, Operator*> operators;


   protected:

        /**
         * @brief fill the operators QMap
         *
         * @return a filled QMap containing all operators
         */
        static QMap<int, Operator *> initializeOperators();
```

```
    public:

        /**
         * @brief predecence accessor
         *
         * @return predecence
         */
        int getPrecedence();

        /**
         * @brief associativity accessor
         *
         * @return associativity type
         */
        Associativity getAssociativity();

        int precedence;
        Associativity associativity;
};

#endif // OPERATOR_H
```

Source A.22 – lib/parser.h

```
#ifndef PARSER_H
#define PARSER_H

#include <QString>
#include <QList>
#include <QObject>
#include <QDebug>
#include <QList>

#include "lib/lexer.h"
#include "lib/token.h"
#include "lib/expressionNode.h"
#include "lib/assignationNode.h"
#include "lib/varNode.h"
#include "lib/calculable.h"


class VarNode;

class Parser : public QObject
{
    Q_OBJECT

    public:

        /**
         * @brief constructor
         *
         * @param source QString to parse
         */
        Parser(QString source);

        /**
         * @brief destructor
         */
        ~Parser();

        /**
         * @brief run the parser
         *
         * @return a Calculable
         */
        Calculable* run();


    public:

        /**
         * @brief isFunction check if a token is a function
         * @param token to test
         * @return type of token
         */
        static bool isFunction(Token token);
```

```
    private:

        /**
         * @brief generate a Node tree representation of a tokenList
         *
         * @param tokens the tokenList to parse
         * @return a Node tree representation of the tokenlist
         */
        Node* generateTree(QList<Token> tokens);

        QString source;
        Lexer *lexer;


    signals:
        void parenthesisError();
};

#endif // PARSER_H
```

Source A.23 – lib/expressionNode.cpp

```
#include "expressionNode.h"
#include "parser.h"


ExpressionNode::ExpressionNode(QList<Token> expression)
{
    this->expression = expression;
    this->value = NULL;

    convertToRPN();
}


ExpressionNode::~ExpressionNode() {}


Calculable* ExpressionNode::execute()
{
    QList<Calculable*> stack;
    QList<Token> expression = this->expression;

    foreach (Token token, expression)
    {
        TokenKind kind = token.getKind();

        if (kind == T_STRING)
        {
            if (Parser::isFunction(token))
            {
                //det(Matrix) : Determinant
                if(token.getValue().toLower() == "det")
                {
                    if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
                        T_MATRIX)
                    {
                        //Get params
                        Matrix* mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
                        stack.removeLast();
                        //Calculate
                        stack.append(MatrixLib::determinant(mat));
                    }
                    else
                    {
                        throw std::runtime_error("Help : det(<b>Matrix</b> mat)");
                    }
                }
                //cof(Matrix mat, Scalar i, Scalar j) : Cofactor
                else if(token.getValue().toLower() == "cof")
                {
                    if(stack.length() >= 3
                            && stack[stack.length()-1]->getType() == T_SCALAR
                            && stack[stack.length()-2]->getType() == T_SCALAR
                            && stack[stack.length()-3]->getType() == T_MATRIX)
                    {
                        //Get params
```

```
            Scalar* j = dynamic_cast<Scalar*>(stack[stack.length()-1]);
            stack.removeLast();
            Scalar* i = dynamic_cast<Scalar*>(stack[stack.length()-1]);
            stack.removeLast();
            Matrix* mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
            stack.removeLast();

            //Checks
            if(i->getRawValue() < 1 || i->getRawValue() > mat->getM())
                throw std::runtime_error("Help : cof(matrix mat, scalar i,
                    scalar j)<br/> And 1 &le; i &le; " +
                    QString::number(mat->getM()).toStdString());
            else if(j->getRawValue() < 1 || j->getRawValue() >
                mat->getN())
                throw std::runtime_error("Help : cof(matrix mat, scalar i,
                    scalar j)<br/> And 1 &le; j &le; " +
                    QString::number(mat->getN()).toStdString());

            //Calculate
            stack.append(MatrixLib::cofactor(mat, i->getRawValue()-1,
                j->getRawValue()-1));
        }
        else
        {
            throw std::runtime_error("Help : cof(<b>Matrix</b> mat,
                <b>Scalar</b> i, <b>Scalar</b> j)");
        }
    }
    //trans(Matrix) : Transpose
    else if(token.getValue().toLower() == "trans")
    {
        if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
            T_MATRIX)
        {
            //Get params
            Matrix* mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
            stack.removeLast();
            //Calculate
            stack.append(MatrixLib::transpose(mat));
        }
        else
        {
            throw std::runtime_error("Help : det(<b>Matrix</b> mat)");
        }
    }
    //co(Matrix) : Cofactor matrix
    else if(token.getValue().toLower() == "co")
    {
        if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
            T_MATRIX)
        {
            //Get params
            Matrix* mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
            stack.removeLast();
            //Calculate
            stack.append(MatrixLib::coMatrix(mat));
```

```
    }
    else
    {
        throw std::runtime_error("Help : co(<b>Matrix</b> mat)");
    }
}
//I(Scalar n) : Generate the identity matrix of size n
else if(token.getValue().toLower() == "i")
{
    if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
        T_SCALAR)
    {
        //Get params
        Scalar* n = dynamic_cast<Scalar*>(stack[stack.length()-1]);
        stack.removeLast();
        //Calculate
        stack.append(MatrixLib::identity(n->getRawValue()));
    }
    else
    {
        throw std::runtime_error("Help : I(<b>Scalar</b> n)");
    }
}
//inv(Matrix mat) : Generate the inverted matrix of mat
else if(token.getValue().toLower() == "inv")
{
    if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
        T_MATRIX)
    {
        //Get params
        Matrix *mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
        stack.removeLast();
        //Calculate
        stack.append(MatrixLib::inv(mat));
    }
    else
    {
        throw std::runtime_error("Help : inv(<b>Matrix</b> mat)");
    }
}
//trace(Matrix mat) : Generate the trace matrix of mat
else if(token.getValue().toLower() == "trace")
{
    if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
        T_MATRIX)
    {
        //Get params
        Matrix *mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
        stack.removeLast();
        //Calculate
        Scalar *tmp = new Scalar(MatrixLib::trace(mat));
        stack.append(tmp);
    }
    else
    {
        throw std::runtime_error("Help : trace(<b>Matrix</b> mat)");
```

```
    }
}
//norm(Matrix mat) : Calculate the norm of a column or row matrix
else if(token.getValue().toLower() == "norm")
{
    if(stack.length() >= 1 && stack[stack.length()-1]->getType() ==
        T_MATRIX)
    {
        //Get params
        Matrix *mat = dynamic_cast<Matrix*>(stack[stack.length()-1]);
        stack.removeLast();
        //Calculate
        Scalar *tmp = new Scalar(MatrixLib::norm(mat));
        stack.append(tmp);
    }
    else
    {
        throw std::runtime_error("Help : norm(<b>Matrix</b> mat)");
    }
}
//solve(Matrix|Scalar A, Matrix|Scalar B) : Solve a linear equation
    of type A*x=B
else if(token.getValue().toLower() == "solve")
{
    if(stack.length() >= 2
           && ((stack[stack.length()-1]->getType() == T_MATRIX &&
               stack[stack.length()-2]->getType() == T_MATRIX)
           || (stack[stack.length()-1]->getType() == T_SCALAR &&
               stack[stack.length()-2]->getType() == T_SCALAR)))
    {
        //Scalar
        if(stack[stack.length()-1]->getType() == T_SCALAR)
        {
            //Get params
            Calculable &B = *(stack[stack.length()-1]);
            stack.removeLast();
            Calculable &A = *(stack[stack.length()-1]);
            stack.removeLast();
            //Result
            stack.append(A/B);
        }
        else
        {
            //Get params
            Calculable &B = *(stack[stack.length()-1]);
            stack.removeLast();
            //Get params
            Matrix *A = dynamic_cast<Matrix*>(stack[stack.length()-1]);
            stack.removeLast();
            //Result
            stack.append(B*(*MatrixLib::inv(A)));

        }
    }
    else
    {
```

```
                throw std::runtime_error("Help : Solve a linear equation of
                    type A*x=B<br/>solve(<b>Matrix</b> A, <b>Matrix</b>
                    B)<br/>solve(<b>Scalar</b> A, <b>Scalar</b> B)");
            }
        }
    }
    else
    {
        VarNode* var = VarNode::getVar(token.getValue());

        if (var->getValue() == NULL)
        {
            throw std::runtime_error("Var " + token.getValue().toStdString()
                + " is not defined.");
            return NULL;
        }

        stack.append(var->getValue());
    }
}
else if (kind == T_SCALAR)
{
    stack.append(new Scalar(token.getValue()));
}
else if (kind == T_MATRIX)
{
    stack.append(new Matrix(token.getValue()));
}
else if (Operator::isOperator(token))
{
    if (stack.length() < 2)
    {
        throw std::runtime_error("Not enough argument for operator " +
            token.getValue().toStdString());
        return NULL;
    }

    Calculable &a = *(stack[stack.length()-1]);
    stack.removeLast();

    Calculable &b = *(stack[stack.length()-1]);
    stack.removeLast();

    switch (token.getKind())
    {
        case T_SUM:
            stack.append(a+b);
            break;

        case T_SUB:
            stack.append(a-b);
            break;

        case T_MULTIPLY:
            stack.append(a*b);
            break;
```

```
                case T_DIVIDE:
                    stack.append(a/b);
                    break;

                case T_MODULO:
                    stack.append(a%b);
                    break;

                case T_POW:
                    stack.append(a^b);
                    break;
                default:
                    throw std::runtime_error("An operator doesn't have its
                        operation");
            }
        }
    }

    return stack[0];
}


QString ExpressionNode::toString() const
{
    if (this->value == NULL)
    {
        return "";
    }

    return "Expression node value: " + this->value->toString();
}


void ExpressionNode::convertToRPN()
{
    //We search for negatives numbers i.e "T_SUB" not used as operator
    for (int i = 0; i < this->expression.length(); i++)
    {
        if (this->expression[i].getKind() == T_SUB)
        {
            //Beginning of expression || parenthesis to the left
            if (i == 0 || this->expression[i-1].getKind() == T_PARENTHESIS_LEFT)
            {
                TokenKind rightTokenKind = this->expression[i+1].getKind();

                //Scalar or string to the right
                if (rightTokenKind == T_SCALAR || rightTokenKind == T_STRING) // ||
                    T_MATRIX ? => TODO T_CALCULABLE needs to be implemented
                {
                    QList<Token> newExpression;

                    //Add "-1 *"
                    newExpression.append(this->expression.mid(0, i));
                    newExpression.append(Token(T_SCALAR, "-1"));
                    newExpression.append(Token(T_MULTIPLY, "*"));
```

```
                newExpression.append(this->expression.mid(i+1,
                    this->expression.length()));
                this->expression = newExpression;
            }
        }
    }
}


QMap<int, Operator*> operators = Operator::operators;
QList<Token> newExpression;
QList<Token> stack;

foreach (Token token, this->expression)
{
    TokenKind kind = token.getKind();

    if (kind == T_SCALAR || kind == T_MATRIX)
    {
        newExpression.append(token);
    }
    else if (kind == T_STRING)
    {
        if (Parser::isFunction(token))
        {
            stack.append(token);
        }
        else
        {
            newExpression.append(token);
        }
    }
    else if (kind == T_COMMA)
    {
        while (stack.length () != 0 && stack[stack.length()-1].getKind() !=
            T_PARENTHESIS_LEFT)
        {
            newExpression.append(stack[stack.length()-1]);
            stack.removeLast();
        }

        if (stack.length() == 0)
        {
            throw std::runtime_error("Parenthesis or comma error");
            return;
        }
    }
    else if (Operator::isOperator(token))
    {
        Operator* currentOperator = operators[token.getKind()];

        while (stack.length() != 0 &&
            Operator::isOperator(stack[stack.length()-1]))
        {
            Operator* stackOperator =
                operators[stack[stack.length()-1].getKind()];
```

```
            if ((currentOperator->getAssociativity() == LEFT_ASSOC &&
               currentOperator->getPrecedence() - stackOperator->getPrecedence()
                   <= 0) ||
               (currentOperator->getAssociativity() == RIGHT_ASSOC &&
               currentOperator->getPrecedence() - stackOperator->getPrecedence()
                   < 0))
            {
                newExpression.append(stack[stack.length()-1]);
                stack.removeLast();
                continue;
            }

            break;
        }

        stack.append(token);
    }
    else if (token.getKind() == T_PARENTHESIS_LEFT)
    {
        stack.append(token);
    }
    else if (token.getKind() == T_PARENTHESIS_RIGHT)
    {
        while (stack.length() != 0 && stack[stack.length()-1].getKind() !=
            T_PARENTHESIS_LEFT)
        {
            newExpression.append(stack[stack.length()-1]);
            stack.removeLast();
        }

        if (stack.length() == 0)
        {
            throw std::runtime_error("Parenthesis error");
            return;
        }

        stack.removeLast();

        if (stack.length() != 0 && stack[stack.length()-1].getKind() == T_STRING
            && Parser::isFunction(stack[stack.length()-1]))
        {
            newExpression.append(stack[stack.length()-1]);
            stack.removeLast();
        }
    }
    else
    {
        newExpression.append(token);
    }
}

while (stack.length() != 0)
{
    newExpression.append(stack[stack.length()-1]);
    stack.removeLast();
```

```
    }

    this->expression = newExpression;
}
```

Source A.24 – lib/operator.cpp

```
#include "lib/operator.h"


Operator::Operator(int precedence, Associativity associativity)
{
    this->precedence = precedence;
    this->associativity = associativity;
}


Operator::~Operator()
{
}


int Operator::getPrecedence()
{
    return this->precedence;
}


Associativity Operator::getAssociativity()
{
    return this->associativity;
}


QMap<int, Operator*> Operator::initializeOperators()
{
    QMap<int, Operator*> operators;

    operators.insert(T_SUM, new Operator(1, LEFT_ASSOC));
    operators.insert(T_SUB, new Operator(1, LEFT_ASSOC));
    operators.insert(T_MULTIPLY, new Operator(5, LEFT_ASSOC));
    operators.insert(T_DIVIDE, new Operator(5, LEFT_ASSOC));
    operators.insert(T_MODULO, new Operator(5, LEFT_ASSOC));
    operators.insert(T_POW, new Operator(10, LEFT_ASSOC));

    return operators;
}


bool Operator::isOperator(Token token)
{
    return operators.keys().contains(token.getKind());
}


QMap<int, Operator*> Operator::operators = Operator::initializeOperators();
```

Source A.25 – lib/assignationNode.h

```
#ifndef ASSIGNATIONNODE_H
#define ASSIGNATIONNODE_H

#include <QDebug>
#include <QString>
#include <QList>
#include <QMap>

#include "lib/node.h"
#include "lib/varNode.h"
#include "lib/expressionNode.h"
#include "lib/token.h"

class ExpressionNode;
class VarNode;

class AssignationNode : public Node
{
    public:

    /**
     * @brief constructor
     *
     * @param variable VarNode to set
     * @param expression expression to execute
     */
    AssignationNode(VarNode *variable, ExpressionNode *expression);

    /**
     * @brief destructor
     */
    ~AssignationNode();

    /**
     * @brief put expression value in memory as varName
     * @return expression value
     */
    virtual Calculable* execute();

    /**
     * @brief variable Node accessor
     *
     * @return a pointer to the variable Node associated to this assocation Node
     */
    VarNode* getVariable() const;

    /**
     * @brief expression Node accessor
     * @return a pointer to the expression Node associated to this association Node
     */
    ExpressionNode* getExpression() const;

    /**
     * @brief toString method
     *
```

```
     * @return a QString representation of the Node
     */
    virtual QString toString() const;


    protected:
        VarNode *variable;
        ExpressionNode *expression;
};

#endif // ASSIGNATIONNODE_H
```

Source A.26 – lib/expressionNode.h

```
#ifndef EXPRESSIONNODE_H
#define EXPRESSIONNODE_H

#include <QDebug>
#include <QString>
#include <QList>
#include <QMap>

#include "lib/node.h"
#include "lib/calculable.h"
#include "lib/token.h"
#include "lib/operator.h"
#include "lib/calculables/scalar.h"
#include "lib/calculables/matrix.h"
#include "lib/matrixlib.h"


class VarNode;

class ExpressionNode : public Node
{
    public:

        /**
         * @brief constructor
         *
         * @param expression the expression in tokens
         */
        ExpressionNode(QList<Token> expression);

        /**
         * @brief destructor
         */
        ~ExpressionNode();

        /**
         * @brief Execute the node
         *
         * @return a Calculable pointer
         */
        Calculable* execute();

        /**
         * @brief toString method
         *
         * @return a QString representation of the Node
         */
        QString toString() const;


    protected:

        /**
         * @brief Convert current expression to a RPN notation
         */
```

```
        void convertToRPN();

        /**
         * @brief Determine if a token is a function or a variable
         *
         * @param token token
         * @return nature of the token
         */
        bool isFunction(Token token);

        QList<Token> expression;
        Calculable* value;
};

#endif // EXPRESSIONNODE_H
```

Source A.27 – lib/varNode.h

```
#ifndef VARNODE_H
#define VARNODE_H

#include <QDebug>
#include <QString>
#include <QList>
#include <QMap>

#include "lib/node.h"
#include "lib/token.h"
#include "lib/expressionNode.h"


class VarNode : public Node
{
    public:

        /**
         * @brief constructor
         *
         * @param varName QString of the varName
         * @param value value
         */
        VarNode(QString varName, Calculable *value);

        /**
         * @brief destructor
         */
        ~VarNode();


    public:

        /**
         * @brief return a var if it exists or a new one if it doesn't
         *
         * @param reference var name
         * @param registry memory where to find/create the var
         *
         * @return a pointer to the new/already existing VarNode
         */
        static VarNode* getVar(QString reference);

        /**
         * @brief global registry accessor
         * @return the global registry
         */
        static QList<VarNode *>* getRegistry();

    private:

        /**
         * @brief Init VarNode::registry
         */
        static QList<VarNode *>* initializeRegistry();
```

```
        static QList<VarNode *>* registry;


    public:

        /**
         * @brief Execute the node
         *
         * @return a Calculable pointer
         */
        virtual Calculable* execute();

        /**
         * @brief toString method
         * @return a QString representation of the Node
         */
        virtual QString toString() const;


        /**
         * @brief Set a value in the registry
         *
         * @param value Calculable to set
         */
        void setValue(Calculable *value);

        /**
         * @brief value accessor
         *
         * @return a pointer to the node's value
         */
        Calculable* getValue() const;

        /**
         * @brief varName accessor
         * @return the var name
         */
        QString getName() const;


    protected:
        QString varName;
        Calculable *value;
};

#endif // VARNODE_H
```

Source A.28 – lib/lexer.h

```
#ifndef LEXER_H
#define LEXER_H

#include <QList>
#include <QString>
#include <QMap>
#include <QRegExp>
#include <QDebug>
#include <QObject>
#include <stdexcept>

#include "lib/token.h"


class Lexer : public QObject
{
    Q_OBJECT

    public:

        /**
         * @brief constructor
         *
         * @param source String going to be tokenized
         */
        Lexer(QString source);

        /**
         * @brief destructor
         */
        ~Lexer();

        /**
         * @brief tokenize the source
         * @return List of tokens
         */
        QList<Token> run();


    protected:

        /**
         * @brief Initialize Lexer::tokens static map
         * @return Map of TokenKind and it's associated regex
         */
        static QMap<TokenKind, QRegExp> initializeTokens();
        static QMap<TokenKind, QRegExp> tokens;


    protected:

        /**
         * @brief try to find a token on a string, begenning on given offset
         *
         * @param line QString source
```

```
         * @param offset index where to begin the search
         *
         * @return Token
         */
        Token match(QString line, int offset);

        QString source;


    signals:
        void lexerError(QString line, int offset);
};

#endif // LEXER_H
```

Source A.29 – lib/node.cpp

```
#include "node.h"

Node::Node() {}

Node::~Node() {}

Calculable* Node::execute(){ return NULL; }

QString Node::toString() const { return ""; }
```