**L’oggetto funzione CLineCalc**

# Informazioni generali

**Obiettivo:**

fornire la possibilità di interpretare algebricamente stringhe.

Le stringhe devono poter contenere:

* gli **operatori binari** ^; \*, /; +, -, con le relative priorità
* gli **operatori unari** ‘+’ e ‘-‘
* le normali chiamate a **funzione** del tipo sin(), cos(), exp() ecc
* **costanti** esplicite eventualmente dotate di esponente con o senza segno
* **variabili** letterali i cui valori sono noti all’esterno e di cui si passano puntatori.

## Requisito:

Siccome l’oggetto deve consentire l’elaborazione algebrica di variabili contenenti anche molti punti, deve essere ottimizzata in termini di efficienza.

Per perseguire questo requisito si opera come segue:

* in una fase A viene acquisita la linea e la relativa lista di variabili e fatta una semplice diagnostica. In particolare, andrà verificato che le variabili non coincidono con i nomi di funzione
* in una fase B la linea sotto esame viene analizzata e sostituita con puntatori a costanti, variabili, funzioni, operatori
* nella successiva fase C la linea contenente solamente puntatori e parentesi viene iterativamente utilizzata per effettuare i calcoli

Ovviamente è la fase C a richiedere la massima efficienza, perché viene richiamata tante volte quanti sono i campioni delle variabili.

## Utilizzo:

0) dati delle variabili esterne sono contenuti nelle righe della matrice \*\*y\_.

1) si passa la stringa (QString) da analizzare attraverso "getLine". Se la linea contiene delle variabili va usata la versione overload di getLine che consente di passare anche i nomi delle variabili presenti nella riga e puntatori ai corrispondenti valori GETLINE FA FASI A E B

2) con i che va da 0 a size-1 si fa result[i]=compute(i); COMPUTE FA LA FASE C

=> eventuali messaggi di errore si trovano nella QString, interna ma pubblica, di nome "err"

## Specifiche per l’utente:

* i nomi di variabile possono contenere solo lettere e numeri e devono cominciare per numero
* I nomi di funzione devono essere immediatamente seguiti dal carattere ‘(‘.

# Funzionamento interno

## Scelta di base

I puntatori a costanti, variabili, funzioni, operatori sono vettori di dimensione pari al numero di caratteri della stringa passata. Questo crea un piccolo spreco di memoria, ma semplifica molto la gestione della linea: l’indice del carattere diventa l’indice del puntatore.

In line dove vi sono i quattro elementi chiave (costanti, variabili, nomi di funzione, operatori binari), vengono messi dei caratteri speciali, che danno la posizione dell’elemento nella linea, e costituiscono l’indice del rispettivo puntatore:

* carattere ‘#’ per le costanti esplicite
* carattere ‘@’ per le variabili
* carattere ‘ù’ per gli operatori unari
* carattere ‘&’ per i nomi di funzione.
* Carattere ‘+’ per gli operatori binari (cioè quelli della stringa “^\*/+-”)

### Nota sugli unari

Gli operatori ‘+’ e ‘-‘ unari sono trattati come segue:

* Gli unari davanti alle costanti esplicite vengono immediatamente attribuiti alla costante all’interno di substPointersToConsts()
* Gli unari davanti alle variabili e alle parentesi vengono trattati in subPointersToOpers(). Ogni unario viene prima riconosciuto. Se è un positivo, il corrispondente carattere viene semplicemente convertito in ‘ ‘. Se è negativo, viene sostituito con il simbolo di operatore unario negativo, ‘-’. La sua elaborazione avviene in computeUnaryMinus() con il livello di precedenza che spetta agli unari secondo le regole del linguaggio C

## Fasi esecutive

In conseguenza della scelta di base di cui al paragrafo precedente le fasi di attribuzione dei puntatori (da B1 a B5) sono le seguenti:

B1 si manda in esecuzione substPointersToConst(). Questa funzione analizza la stringa ed identifica tutti i numeri espliciti. Essi vengono sostituiti con il carattere '#'. La posizione di tale carattere costituisce l'indice di un puntatore al numero sostituito

B2 si manda in esecuzione substPointersToFuns(). Questa funzione identifica le funzioni presenti e sostituisce ad esse il '&'. Questo è possibile perché quando viene mandata in esecuzione questa funzione la stringa line non contiene più alcun carattere letterale.

B3 si manda in esecuzione getVarPointers(). Con questa funzione si passano tutti i nomi delle variabili. Durante l'esecuzione di questa routine internamente i corrispondenti nomi in line vengono sostituiti con il carattere '@'. La posizione di tale carattere costituisce l'indice di un puntatore al numero sostituito. Tale puntatore viene richiesto all'esterno attraverso un signal.

B4 si manda in esecuzione simplify() che fa operazioni di ottimizzazione. L’unica operazione attualmente in programma è l’eliminazione delle coppie di parentesi racchiudenti un unico valore (di tipo ‘#’ o ‘@’). Questo può essere frequente se si usano le funzioni: ad es. sin(x) prima di simplify() diviene “& (x).

B5 si manda in esecuzione substPointersToOpers(). In questa funzione tutti gli operatori binari vengono sostituiti con il carattere ‘+’ mentre l’informazione relativa all’operatore originariamente presente in quella posizione è memorizzata in pOper[]. Si ricorda che i valori di pOper sono 0, 1, 2, 3, 4 rispettivamente per ^, \*, /. +, -.

NOTA

La sequenza delle precedenti operazioni è importante.

Ad esempio non si può scambiare la B2 con la B3. Se infatti ad esempio ho una variabile di nome “s” se scambiassi le due fasi sarebbe complicato distinguere “s” del nome di variabile dalla “s” prima lettera di sin;()

FASI DI CALCOLO C0 A C4: EFFETTUAZIONE OPERAZIONI IN ORDINE DI PRIORITA' quando si fa compute()

C0) Si gestiscono le parentesi che hanno la funzione di alterare le priorità delle espressioni di effettuano le chiamate a funzione: viene definita una sottostringa con il contenuto della parentesi, e con tale contenuto viene effettuata una chiamata ricorsiva alla funzione di calcolo compute()

C1) Si effettuano le chiamate a funzione. Le funzioni sono caratterizzate da una 'f' a destra della quale è presente, con l'interposizione al più di qualche spazio, un '#' o un '@'

C2-C4) si analizzano gli operatori con le rispettive priorità. Gli operatori disponibili, in ordine di priorità decrescente sono i seguenti:

- '^' (da SX a SX)

- '/' e '\*' binari (da SX a DX)

- '+' e '-' binari (da SX a DX

**Alla fine del processo la stringa conterrà un unico carattere '#', la cui posizione è l'indice di un puntatore al risultato**

# Estensione per l’uno in PlotXY

La funzione getLine ha la seguente signature:

QString CLineCalc::getLine(QByteArray line\_, QList <QByteArray> nameList\_, float \*\* y\_){

Richiede quindi la lista dei nomi presenti in line\_ e la corrispondente matrice di valori.

Quindi occorre una funzione di preparazione che crei lista e matrice a partire dai dati iniziali, che possono fare riferimento anche a differenti files.

## Generazione argomenti per getLine

Devo fare una funzione

struct SInputData {QList <QByteArray> nameList; float \*\* y\_;};

struct SInputData CLineCalc::XYPreprocess (fileNumsLst, varNumsLst)

Che è specifica per l’uso di XY, che costruisca gli argomenti da passare a getLine().

Naturalmente per fare questa funzione mi posso ispirare all’attuale codice di lineCalc() in PlotXY.

Occorre ricordare che VarTable() già compila funInfo quando viene inserita la stringa. Questa struttura contiene quello che serve:

QList <SXYNameData> funInfo

struct SXYNameData{

bool allLegalNames; //true se tutte le variabili contengono un nome legale Se allLegalNames=false il contenuto delle altre variabili della struttura è indeterminato in quanto alla prima variabile non valida l'analisi della stringa di input è interrotta

bool rightScale; //dice se la variabile va plottata verso l'asse verticale destro o no.

bool integralRequest; //dice se si sta richiedendo l'integrale di una stringa

QColor color;

QList <int> fileNums; //contiene la lista dei numeri dei files da cui è necessario prelevare le variabili della stringa (un item in lista per ogni file differente)

QList <SVarNums> varNumsLst; //Numeri delle variabili. Un item in lista per ogni variabile differente

QList <QString> varNames; //i nomi delle variabili della stringa. Forniscono un'informazione meno sofisticata di varNumsLst ma utile quando non serve la scomposizione di dettaglio ma solo i nomi.

QList <QString> varUnits; //i nomi delle variabili unità di misura delle variabili della stringa. L'ordine è il medesimo di varNames.

QString line; //v. CLineCalcDevel

QString lineInt; //v. CLineCalcDevel

QString name; //nome attribuito alla funzione di variabile descritta.

QString ret; //messaggio di errore compilato se allLegalNames=false;

};

Quindi la chiamata a XY preprocess sarà:

struct SInputData CLineCalc::XYPreprocess (funInfo)

## Codice per la generazione di funInfo da parte di VarTable()

VarTable usa checkAndfindNames() e getNameAndMatrix() di CLineCalc di PlotXY.

Esse vanno quindi mantenute, ed eventualmente adattate e semplificate.

## Limitazione del numero di funzioni possibili

Vorrei limitare le funzioni possibile a

sin, cos, exp, abs, sqrt.

Posso farlo in maniera flessibile lasciando attive tutte le 13 funzioni, ma definendo un vettore di puntatori alle funzioni utilizzabili:

usableFunctions[]

oppure una lista di interi

QList <int>usableFunctions;

Nel caso di PlotXY avrà dimensione 5 e i suoi valori punteranno alle relative funzioni nel menù delle 13 disponibili. All’avviamento CLineCalc inizializzerà questo vettore (o questa lista) in modo che tutte e 13 le funzioni saranno disponibili in assenza di cambiamenti da parte di utente.

Se però l’utente usa

setUsableFunctions (QList <int> funNums)

con essa può fare l’override delle funzioni utilizzabili default (che, come detto, sono tutte).

## Uso dell’integrale

CLineCalc di funPlot non ha il calcolo dell’integrale, il cui codice va cuqindi estrapolato da analogo codice di CLineCalc di PlotXY

## Uso delle unità di misura

CLineCalc di funPlot non ha il calcolo delle unità di misura, il cui codice va quindi estrapolato da analogo codice di CLineCalc di PlotXY

C’è la funzione computeUnits() che è chiamata direttamente da CDataSelWin.