Математички факултет
Универзитет у Београду

Развој библиотеке математичких функција за пројекат отвореног кода *QLab*

Мастер рад

Дарко Павловић
Октобар 2015

Ментор:
Др Мирослав Марић

Садржај

1. Увод

Qlab је апликација отвореног кода, која се користи за нумеричка израчунавања. Пројекат је покренут на Математичком факултету Универзитета у Београду и од самог почетка апликација је замишљена као бесплатна алтернатива комерцијалном алату *MATLAB*. Израђује се у најновијим технологијама компаније *Microsoft*, као што су *.Net Framework 4* и *WPF*.

Пројекат је у целости прилично компликован, што је условило поделу у неколико потпројеката. У овом раду је описана имплементација математичких функција у *QLab*-у, као и повезаност тог дела пројекта са осталим потпројектима.

1.1. Идеја пројекта *QLab*

1.1.1. Математички алат *MATLAB*

MATLAB представља интерактивно окружење и програмски језик, чији је развој условљен потребама људи који се у свом послу често сусрећу са разним захтевним нумеричким израчунавањима. С обзиром на многе могућности које су уграђене у окружење и програмски језик, *MATLAB* је тренутни лидер на тржишту алата сличних намена. На званичном сајту компаније која развија *MATLAB* (компанија *MathWorks*) доступан је податак да *MATLAB* користи неколико милиона корисника широм света[1].

Име *MATLAB* је скраћеница настала од енглеске фразе *Matrix Laboratory*, која у преводу значи ,,Лабораторија матрица“. У почетку, алат је представљао омотач око библиотека *LINPACK* и *EISPACK* написаних у *Fortran-у*, које су се користиле за израчунавања у линеарној алгебри. Идеја је била да се студентима обезбеди коришћење могућности које пружају те библиотеке, а да не морају претходно да науче да програмирају у *Fortran*-у. Са временом је алат надограђиван новим функционалностима које су омогућиле да се *MATLAB* користи у различите сврхе. Тренутно се *MATLAB* користи у кибернетици, аеро и свемирској индустрији, телекомуникацијама, машинској индустрији и индустрији производње медицинских апарата. Уз коришћење још неких алата истих аутора, *MATLAB* често представља први избор људи који се баве разним истраживањима техничке природе у оквиру факултета и института. Неретко се користи за математичке

[1] Страница на којој се може пронаћи податак доступна је на адреси http://www.mathworks.com/company/aboutus/.

симулације различитих система. Више од 5000 универзитета широм света лиценцирало је своју верзију *MATLAB*-а како би могли да је користе у претходно поменуте сврхе[2].

1.1.2. Алтернативе *MATLAB*-у

MATLAB се тренутно развија као комерцијална апликација. То значи да свако ко жели да користи овај производ мора да лиценцира своју верзију, што чини алат недоступним одређеном броју људи којима је потребан. Из тог разлога, кроз историју су људи покушавали да направе бесплатну алтернативу. У наредном делу ће бити речи о најпознатијим постојећим алтернативама *MATLAB*-а.

GNU Octave

GNU Octave је алат намењен нумеричким израчунавањима. Пројекат је започет на Универзитету у Орегону, за потребе аутора који се у то време бавио инжењерингом хемијских реакција. Било је предвиђено да се користи у паралели са књигом за предавања и основна намена му је била решавање проблема везаних за хемијске реакције. 1993. године у потпуности је написан од нуле како би могао да се користи и за проблеме друге природе. Израђен је у програмском језику *C++* и користи *LAPACK* и *BLAS* библиотеке за разна израчунавања (као и *MATLAB*). Синтакса је прилично слична *MATLAB*-овој, али не и идентична, па је у неким случајевима потребно урадити адаптацију кода да би се променило окружење. Алат је у потпуности бесплатан и може се покренути на оперативним системима *Windows*, *Mac OS* и *Linux*.

FreeMat

FreeMat је окружење које је покренуто са идејом да буде компатибилно *MATLAB*-у и алату *GNU Octave*. Доступно је на оперативним системима *Windows*, *Mac OS* и *Linux*. Међутим, овај алат нема заједницу људи који активно доприносе пројекту и ни приближно толико функционалности као *MATLAB*.

---

[2] Страница на којој се може пронаћи податак доступна је на адреси http://www.mathworks.com/company/aboutus/.

Scilab

Scilab је пројекат отвореног кода, који представља мултиплатформско окружење и виши програмски језик за решавање нумеричких проблема. Развио га је за своје потребе Француски национални институт за истраживања (*INRIA*) 1990. године. На основу званичне веб стране пројекта[3], синтакса програмског језика углавном се заснива на *MATLAB* језику. Нажалост, она није у потпуности иста. Због компатибилности, аутори су у окружење уградили могућност превођења кода са језика *MATLAB* на језик *Scilab*. *Scilab* је такође доступан на оперативним системима *Windows*, *Mac OSX* и *Linux*, а његов код може да се користи и мења бесплатно.

Иако су ови алати прилично добри, њихови аутори кренули су у имплементацију са другим намерама, те они нису у потпуности компатибилни са *MATLAB*-ом. Многе функционалности нису подржане ни у једном од ових алата, па људи који би желели да их користе, морали би најпре да науче нове језике и утроше време да своје претходне пројекте у *MATLAB-у* адаптирају новом окружењу. *GNU Octave* је тренутно, по мишљењу многих, најквалитетнији алат међу наведеним, али лош кориснички интерфејс и слабе перформансе током неких израчунавања на *Windows* оперативном систему сврставају га далеко испод квалитета *MATLAB*-а. Детаљи о упоредној анализи претходно наведених алата могу се пронаћи у [1].

1.1.3. Идеја и циљеви пројекта *QLab*

Имајући у виду велику потребу за бесплатном алтернативом *MATLAB*-а, професори и студенти Математичког Факултета Универзитета у Београду су дошли на идеју да покрену пројекат са циљем да направе своју верзију. Постављени су следећи потциљеви:

- **Потпуна компатибилност са програмским језиком *MATLAB*** - *QLab* би требало да подржи све функционалности које постоје у програмском језику *MATLAB* као и све функционалности које нуди само окружење *MATLAB*.
- **Коришћење најновијих технологија** – пројекат је створио шансу да студенти стекну искуство у технологијама које би касније могли да користе у раду. Такође, идеја је била да *QLab* има најбоље могуће перформансе, да визуелно изгледа добро и да буде једноставан за коришћење а коришћење нових технологија је могло да помогне у томе.
- **Тимски рад** - једна од главних идеја водиља пројекта. Студенти током студија нису имали много прилике да раде у тимском окружењу, па је

[3] Званична страна пројекта налази се на адреси: http://www.scilab.org.

замисао била да они стекну одређено искуство рада у тиму пре него што започну своје каријере.

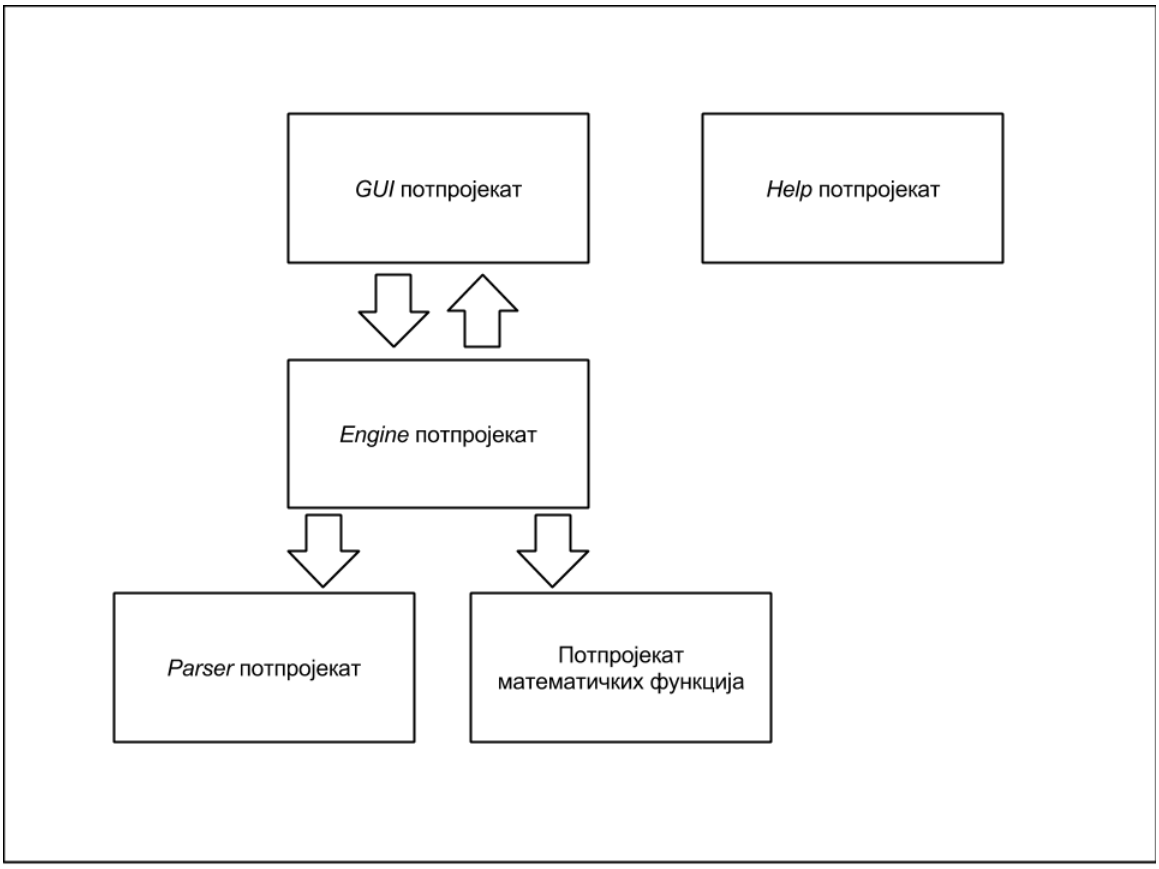
- **Шанса да студенти свих студијских програма учествују у пројекту** – идеја је била да се направи математички софтвер. То је створило могућност да се и студенти других, више математички орјентисаних студијских програма, укључе у развој.

1.2. Организација пројекта *QLab*

Након дефиниције циљева, кренуло се у планирање пројекта. Требало је да *QLab* има неколико функционалности које су могле да се посматрају независно једна од друге. Те функционалности су подељене у потпројекте, како би могле да се реализују паралелно. Тако је остварена следећа подела:

- ***GUI* потпројекат** – целина која би омогућила кориснику да унесе код који жели да покрене. Било је потребно да корисник буде у могућности да код запамти у датотеку и отвори из већ постојеће датотеке. Такође, једна од функционалности овог потпројекта је и презентација резултата извршавања.
- ***Parser* потпројекат** – потпројекат чија је улога да прочита код који је корисник написао и да га трансформише у конструкте које ће *Engine* потпројекат моћи да изврши.
- **Потпројекат математичких функција** – потпројекат који би се бавио имплементацијом функција које се користе за израчунавања.
- ***Engine* потпројекат** - потпројекат који повезује све остале делове. Његов задатак је да након што је покренут програм из *GUI* потпројекта контактира *Parser* потпројекат да преведе код у познате конструкте и да коришћењем функционалности из других делова изврши тај код.
- ***Help* потпројекат** – потпројекат чија је одговорност израда документације и упутстава за коришћење.

Слика 1. илуструје зависности између потпројеката на пројекту *QLab*.



Слика 1. Организација пројекта *QLab*

1.3. Циљеви потпројекта математичких функција

Пре почетка развоја математичких функција у *QLab*-у, једина позната информација била је да *MATLAB* има веома много функција и да је потребно обезбедити све функционалности како би компатибилност била потпуна. У том тренутку постављени су следећи циљеви потпројекта математичких функција:

- **Направити план имплементације функција** - овај циљ бавио се одређивањем најефикаснијег и најбржег начина да се све функције имплементирају. Такође, требало је одредити и редослед имплементације функција.
- **Направити интерфејс који ће омогућити функцијама да једноставно буду позиване из сваког дела пројекта** - постојала је потреба за архитектуралним решењем које би омогућило да комплексност позивања буде на најмањем могућем нивоу.

- **Имплементирати све функције чија је имплементација у том тренутку могућа** - требало је идентификовати функције које не зависе од других делова пројекта који тренутно нису имплементирани и имплементирати те функције.
- **Обезбедити квалитет написаних функција** – требало је направити механизам који проверава да ли имплементиране функције враћају исти резултат као и функције у *MATLAB*-у када су позване са истим параметрима.

Имајући ове циљеве у виду, направљен је план који би омогућио остваривање истих.

1.4. План израде функција

У својој библиотеци, *MATLAB* има преко 10000 функција[4]. Све ове функције имплементиране су у *MATLAB*-у на један од следећих начина: заједно са осталим конструктима и елементима *MATLAB*-а, у програмском језику у којем је написан *MATLAB* (ове функције ће се наставку рада звати уграђеним) или су написане у програмском језику *MATLAB*, где се за њихову имплементацију користе уграђене функције. Статус функције (да ли је функција уграђена или не) може да се добије као излазна вредност функције *feval* програмског језика *MATLAB* када аргумент са којим је функција *feval* позвана представља име те функције. Имајући то у виду, написан је програм који је за сваку функцију у *MATLAB*-у урадио проверу да ли је она уграђена или не. Излаз је била листа која садржи око 450 уграђених функција. План је био да прво буду одрађене све уграђене функције, а затим остале где би за њихову имплементацију биле искоришћене уграђене функције.

Поједине уграђене функције нису могле одмах да буду имплементиране зато што су зависиле од функционалности које, у моменту када је развој отпочео, нису постојале (нпр. *plot* је функција која исцртава график неке математичке функције; она у овом тренутку није могла да се имплементира јер још није почела израда *GUI* потпројекта). Прегледом листе уграђених функција и анализом зависности тих функција од других делова пројекта, добијена је подлиста функција које су одмах могле да се имплементирају. Функције на тој листи су тригонометријске функције, операције са битовима, функције за манипулацију низовима карактера и разне математичке операције. У *QLab*-у су назване математичким функцијама.

[4] Списак је доступан на адреси:
http://www.mathworks.com/help/matlab/functionlist.html.

Направљен је план према ком је редослед имплементације био следећи:

- Све уграђене функције, које могу да се имплементирају, биће написане одмах (листа ових функција биће наведена касније).
- Када буде направљен помак на *GUI* и *Engine* потпројектима, биће настављена имплементација свих уграђених функција које имају зависности од ових пројеката.
- Имплементација осталих функција ће почети након што је завршена имплементација свих уграђених функција.

2. Израда функција

2.1. Изглед математичке функције

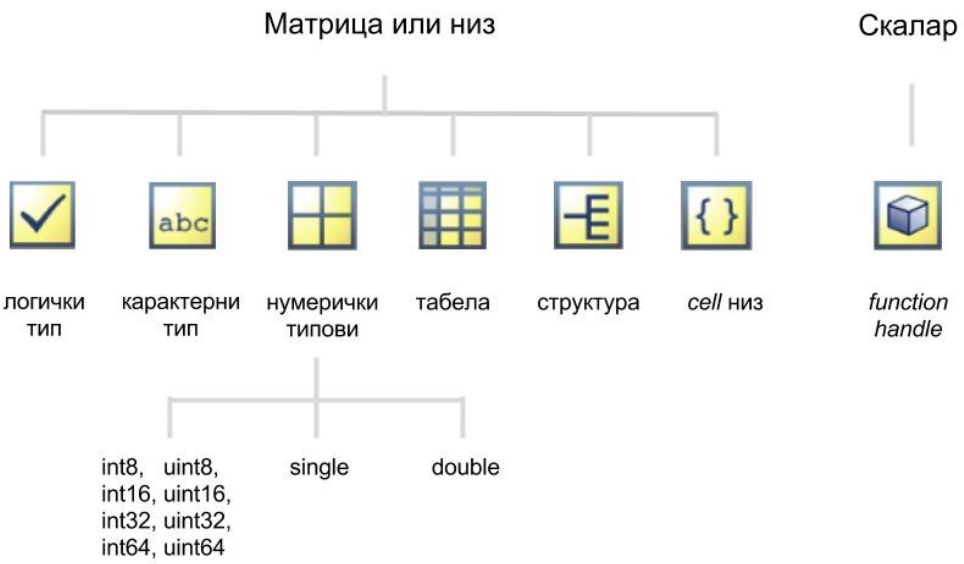
Како би имплементација функција почела, требало је дефинисати изглед функција. Морала је да се донесе одлука како ће изгледати улазни аргументи и излазне вредности функција, како да буду имплементиране функције чији је излаз више од једне повратне вредности и како да буду имплементиране функције које се понашају двојако у зависности од тога колико улазних вредности имају. Да би се дошло до одговора на сва ова питања, неопходно је било анализирати како раде функције у *MATLAB*-у, размислити о томе како су имплементирани типови података у *QLab*-у и како ће се функције позивати од стране *Engine* потпројекта. У наставку овог поглавља је објашњено како је све ово утицало на коначан изглед прототипа функције.

2.1.1. Типови података у MATLAB–у

Једна од ствари која утиче на изглед функције у *QLab*-у јесте структура типова података. Имајући у виду да је крајњи циљ целог пројекта била компатибилност скупа функционалности са *MATLAB*-ом, морала је да буде постигнута и компатибилност функционалности механизама за чување података.

MATLAB има веома велики избор структура података, како би се што боље прилагодио потребама огромног броја корисника. Корисници могу да креирају вишедимензионалне матрице целобројних и бројева у покретном зарезу, карактера и логичних вредности. Такође, постоје структуре које се користе да се вредности различите природе упакују у једну структуру.

Слика 2. приказује основне структуре података у *MATLAB*-у.



Слика 2. Типови података у *MATLAB*-у

Од 16 доступних типова података, сви сем структуре *function handle* имају форму матрице или низа. У случају кад је тип података представљен матрицом, та матрица може имати онолико димензија колико је потребно крајњем кориснику. Тип *function handle* је, са друге стране, представљен само једном вредношћу (појам који се користи у оваквим ситуацијама је скалар).

У *MATLAB*-у постоје два начина за дефиницију матрице. Први начин захтева да се, приликом дефиниције матрице, сваки елемент матрице наведе. Овај начин се назива потпуна дефиниција матрице. Други начин је непотпуна дефиниција матрице (термин који се користи за овако дефинисану матрицу је *sparse* матрица). У том случају се дефинишу само елементи који су различити од нуле и позиције на којима се ти елементи налазе у матрици. У оба случаја, матрица је у меморији представљена на исти начин. Разлика је само у начину на који корисник дефинише објекат. Други начин је погоднији за корисника уколико матрица има мало елемената различитих од нуле. У случају нумеричког, логичког и карактерног типа, елемент унутар матрице који представља вредност неког типа је заправо пар вредности, зато што свака вредност унутар матрице представља комплексан број.

Што се тиче нумеричких типова, они се деле у две подгрупе – подгрупу целобројних бројева и подгрупу бројева са покретним зарезом. Разлика између типова је само у количини меморије коју вредност типа заузима (и самим тим у опсегу вредности које могу да буду сачуване у тој структури података). Вредности у структури бројева са покретним зарезом могу да буду и невалидне (енглески *NaN*)

и бесконачне (енглески *Infinity*) вредности. Целобројне структуре могу да садрже означене и неозначене бројеве. Над свим овим типовима могу да буду извршене све нумеричке операције, као и операције са матрицама.

Карактерни и логички тип имају исти концепт као и нумерички типови. Представљени су матрицом, другачији је само елемент унутар матрице. Карактерни тип се користи за чување низова карактера. Над објектом овог типа су, поред стандардних операција са матрицама, могуће и све операције које су могуће над нискама карактера у свим вишим програмским језицима (надовезивање, тражење подниске итд.). Логички тип чува вредности тачно и нетачно, и над њим су могуће све постојеће логичке операције.

Сви поменути типови се користе за чување низова података истог типа. Са друге стране, табеле и структуре представљају типове који омогућавају кориснику чување података различитих типова унутар једне структуре. Табела је структура у којој се чувају подаци којима је природан табеларни приказ (приказ по колонама). Ова структура је погодна за коришћење када постоји потреба за анализом података који су нпр. доступни у *CSV* формату (видети [2] за више детаља). Структура је низ именованих вредности које могу да буду било које величине или типа. Наликује структурним типовима у свим осталим вишим програмским језицима.

Cell низ је низ вредности које могу да буду било ког типа. Разлика између структуре и *cell* низа је у именовању вредности. Овај детаљ прави разлику приликом приступа вредностима унутар ових структура података – док се у *cell* низу вредностима приступа као елементима у низу, унутар структуре, вредностима се приступа на основу имена.

Function handle је тип који представља референцу на функцију у *MATLAB*-у. Једна од ситуација где је пригодна употреба овог типа података је извршавање различитих функција над истим аргументима где се одабир функције која треба да буде извршена догађа у време извршавања програма и зависи од резултата извршавања. Друга ситуација је када корисник има потребу да користи локалну функцију ван контекста у ком је она креирана. Наиме, дизајн језика *MATLAB* је такав да локална функција може да буде позвана само унутар фајла у коме је дефинисана. Ако се референца на локалну функцију врати као повратна вредност неке глобалне функције, контекст у коме се функција позива може да се прошири и тако ово ограничење језика постаје превазиђено.

2.1.2. Имплементација типова података у *QLab*-у

Класа NDimArray

Најважнији део система за чување података у *QLab*-у је класа *NDimArray*. У наставку поглавља је детаљно објашњена структура те класе.

Класа *NDimArray* је написана са намером да покрије све функционалности вишедимензионалних матрица у *MATLAB*-у (чување података и манипулације као што су промена димензија и додавање друге матрице на крај). Пошто су, при манипулацији са свим типовима, функционалности вишедимензионалних матрица исте, одлучено је да ова структура буде генеричка, како би та логика била искоришћена на више места:

```
public class NDimArray<T> : ICloneable
```

Први захтев који је требало да испуни ова структура је чување података. У ту сврху користе се два низа података. Први низ је низ *_data* у коме се смештају вредности. Свака вредност се смешта у низ на позицију која се израчунава на основу координата где та вредност треба да се налази унутар вишедимензионалне матрице. Друга структура *_dimensionSize* представља димензије вишедимензионалне матрице која је представљена тренутном структуром. Дефиниција тих поља изгледа овако:

```
private T[] _data;
private  int[] _dimensionSize;
```

У наставку су написане поменуте функције које претварају координате у вишедимензионалном низу у позицију у низу за чување података:

```
private int CoordinatesToPosition(params int[] coordinates)
{
      int retval = coordinates[Dimension - 1];

      for (int i = Dimension - 2; i > -1; i--)
            retval = retval * _dimensionSize[i] + coordinates[i];

      return retval;
}

private int[] PositionToCoordinates(int position)
{
      var coordinates = new int[Dimension];
      var rest = position;
```

```
            for (int i = 0; i < Dimension; i++)
            {
                  coordinates[i] = rest % _dimensionSize[i];
                  rest = rest / _dimensionSize[i];
            }

            return coordinates;
      }
```

Притом је у коду искоришћено поље које представља димензију структуре, а добија се као дужина низа у коме се чувају димензије.

Следећи захтев који би требало да класа испуни је једноставно креирање. Класа има три конструктора: подразумевани, који креира празну дводимензионалну матрицу чије су обе димензије 0, затим конструктор који прихвата димензије и прави структуру тих димензија, која има подразумеване вредности и најзад конструктор који осим димензија, прихвата и низ елемената који су поређани у редоследу објашњеном раније.

```
public NDimArray();
public NDimArray(params int[] dimensionSize);
public NDimArray(T[] data, params int[] dimensionSize);
```

На крају, постоји захтев да се омогући директан приступ сваком елементу. То је могуће захваљујући оператору за приступање елементу који је написан. Оператор прима координате у вишедимензионалном низу и враћа елемент на тим координатама. Оператор може да се користи и за читање и за памћење елемента.

```
public T this[params int[] coordinates];
```

Итерација кроз структуру NDimArray

Структура описана у претходном поглављу је у стању да запамти низ елемената и да омогући приступ појединачно сваком од њих.

Међутим, у *MATLAB*-у постоји мноштво функција које захтевају да се на сваку вредност у структури примени одређена трансформација. Зато је, за ову потребу, направљена класа која служи као итератор кроз структуру *NDimArray*. Та класа се зове *Counter*.

Класа има следеће методе:

```
public Counter(int[] size);
public int[] getCurrent();
public void reset();
```

```
public bool isAtEnd();
public void plusPlus();
```

Идеја је да се конструктор позове са улазним вредностима које представљају димензије вишедимензионалног низа који треба да се обиђе. Метода *plusPlus*() повећава бројач, док метода *getCurrent*() враћа као излазне вредности тренутне координате. Метода *isAtEnd*() проверава да ли је бројач стигао до краја, а метода *reset*() се користи да се бројач врати на координате првог елемента.

Пример кода који приказује пролазак кроз вишедимензионални низ коришћењем ове класе изгледа овако:

```
…
Counter c = new Counter(NDimArrayExample.DimensionSize);
for (c.reset(); !c.isAtEnd(); c.plusPlus())
{
      doSomething(NDimArrayExample[c.getCurrent()]);
}
…
```

Коришћењем горњег кода, редослед обиласка елемената на примеру вишедимензионалног низа димензија 2,3,3 би био:

```
0, 0, 0
0, 0, 1
0, 0, 2
0, 1, 0
0, 1, 1
0, 1, 2
0, 2, 0
0, 2, 1
0, 2, 2
1, 0, 0
1, 0, 1
1, 0, 2
1, 1, 0
1, 1, 1
1, 1, 2
1, 2, 0
1, 2, 1
1, 2, 2
```

Klasa AtomicTypeValue и њене поткласе

Са додавањем могућности итерације, представљена структура у потпуности може да задовољи захтеве које носи концепт вишедимензионалног низа. Како би структура била потпуна, последњи корак је био дефинисање имплементације елемента унутар вишедимензионалног низа.

Раније је напоменуто како је вредност у вишедимензионалном низу којим су представљени неки типови у ствари пар вредности и да те вредности представљају комплексни број. Прва вредност у пару је реални део тог комплексног броја, док је друга вредност у пару имагинарни део.

Апстрактна генеричка класа *AtomicTypeValue* написана је како би представљала заједничку логику, која је касније искоришћена за дефиницију вредности које су градивни елементи вишедимензионалних типова у *QLab*-у.

Структура те класе је једноставна. *_real* представља прву поменуту вредност, *_imag* представља другу:

```
protected T _real;
protected T _imag;
```

Вредности могу да се прочитају или поставе коришћењем метода за приступ пољу. Могу, такође, и директно да се поставе из конструктора. Остале методе нису потребне.

Из ове класе даље су изведене следеће класе:

```
public class QBoolean : AtomicTypeValue<bool>
public class QFloat32 : AtomicTypeValue<float>
public class QFloat64 : AtomicTypeValue<double>
public class QInt8 : AtomicTypeValue<short>
public class QInt16 : AtomicTypeValue<short>
public class QInt32 : AtomicTypeValue<int>
public class QInt64 : AtomicTypeValue<long>
public class QUInt8 : AtomicTypeValue<ushort>
public class QUInt16 : AtomicTypeValue<ushort>
public class QUInt32 : AtomicTypeValue<uint>
public class QUInt64 : AtomicTypeValue<ulong>
public class QChar : AtomicTypeValue<char>
```

У те класе су укључени и конструктори који могу да буду позвани са објектима других типова као улазним вредностима, и на тај начин је решен проблем претварања вредности из једног типа у други.

Имплементација типова у QLab-у

Када су дефинисане класа *NDimArray* и поткласе класе *AtomicTypeValue*, тривијално могу да се дефинишу нумерички, карактерни и логички типови података:

```
public class QBooleanNDimArray : NDimArray<QBoolean>
public class QCharNDimArray : NDimArray<QChar>
public class QFloat32NDimArray : NDimArray<QFloat32>
public class QFloat64NDimArray : NDimArray<QFloat64>
public class QInt8NDimArray : NDimArray<QInt8>
public class QInt16NDimArray : NDimArray<QInt16>
public class QInt32NDimArray : NDimArray<QInt32>
public class QInt64NDimArray : NDimArray<QInt64>
public class QUInt8NDimArray : NDimArray<QUInt8>
public class QUInt16NDimArray : NDimArray<QUInt16>
public class QUInt32NDimArray : NDimArray<QUInt32>
public class QUInt64NDimArray : NDimArray<QUInt64>
```

Структуре, табеле, *function handle* тип и *cell* низ, иако на неки начин имају форму низа, нису могли да искористе претходно описану логику. Имплементирани су као самостални типови података. Дефинисани су типови:

```
QStructValueNDimArray
QCell
QFunctionHandle
QTable
```

Ови типови немају додира са математичким функцијама, па у наставку нема превише детаља о имплементацији истих.

Потреба за енумом IQLabValueType и интерфејсом IQLabValue

MATLAB често мора да изврши превођење вредности из једног у други тип, како би дошао до резултата функције. Такође, у многим случајевима дешава се да функција, у зависности од типа и броја аргумената, враћа различите резултате. Да би све функционалности *MATLAB*-а биле имплементиране, требало је да се уради много провера типова. Овакве ситуације су, током имплементације *QLab*-а, могле да буду решене коришћењем оператора *instanceof*, али је, због практичности, одлучено да постоји енум тип који садржи сваки могући тип података.

Дефинисан је енум *IQLabValueType*, чије вредности одговарају свим типовима података у *QLab*-u:

```
public enum IQLabValueType
{
      QCell,
      QStructValueNDimArray,
      QFunctionHandle,
      QTable,
      QBooleanNDimArray,
      QCharNDimArray,
      QFloat32NDimArray,
      QFloat64NDimArray,
      QInt16NDimArray,
      QInt32NDimArray,
      QInt64NDimArray,
      QInt8NDimArray,
      QUInt16NDimArray,
      QUInt32NDimArray,
      QUInt64NDimArray,
      QUInt8NDimArray
 }
```

Дефинисан је, такође, и интерфејс *IQLabValue*, који тренутно од класа које га имплементирају захтева само да имају имплементирану методу који враћа тип:

```
public interface IQLabValue: ICloneable
{
      IQLabValueType GetEnumType();
}
```

Сви типови у *QLab*-у морају да имплементирају овај интерфејс и врате одговарајућу вредност.

2.1.3. Прототип математичке функције

Приликом имплементације типова података, додато је и неколико детаља који су олакшали израду функција. Решен је проблем пребацивања вредности из једног у други тип и дефинисан је начин да се једноставно и читљиво провери тип неког објекта. Дефинисан је, такође, и начин генерализације типова података у *QLab*-у (објекат који имплементира интерфејс *IQLabValue* је објекат једног од типова у *QLab*-у). Следећи корак је био рад на функцијама.

Провером рада функција у *MATLAB*-у, установљено је да могу да буду позване са променљивим бројем улазних вредности и да врате променљив број повратних вредности, у зависности од тих улазних аргумената.

Такође, логично је било да *Engine* потпројекат не мора да израчунава типове са којима позива функције, већ да само проследи оно што је корисник проследио.

Знајући те информације, дефинисан је прототип функције који захтева да улазне вредности са којима је функција позвана буду задате листом елемената који су објекти типова података у *QLab-y* и да функција као излазну вредност врати листу таквих објеката. Прототип функције изгледа овако:

```
public List<IQLabValue> function(List<IQLabValue> listOfArguments);
```

Дакле, свака функција прима листу објеката који имплементирају интерфејс *IQLabValue* као улазни аргумент и враћа листу објеката исте природе.

2.2. Интерфејс за позивање функција

У организацији пројекта *Qlab* идеја је била да се математичке функције позивају из *Engine* потпројекта након што је *Engine* потпројекат препознао конструкте језика коришћењем функционалности *Parser* потпројекта.

У тренутку позивања, *Engine* потпројекат има приступ улазним аргументима функције, као и ниски карактера која представља њено име. Знајући ту информацију, морао је да буде направљен интерфејс који би омогућио да функција може да се позове коришћењем само тих података. Тако се јавила потреба за везом између имена функције и њене имплементације и морале су да буду решене непознанице које се тичу изгледа саме везе и структуре података у којој би се везе чувале.

Структура података која чува везе је морала да задовољи неколико ствари: да уме да сачува све везе између имена и функције, да уме да претражи везе на основу имена и да претрагу ради ефикасно, имајући у виду да може да постоји неколико хиљада функција. Решење које задовољава ове захтеве и које је на крају искоришћено је хеш мапа (на енглеском се, у зависности од контекста и начина имплементације структуре користе термини *HashMap*, *HashTable*, *Map*, *Dictionary*…). Хеш мапа је структура која пружа могућност да парови података буду сачувани и да касније ти подаци могу ефикасно да се претраже. Механизам на основу којег хеш мапа функционише ради тако што на основу првог елемента пара података израчуна адресу у меморији где би требао да буде смештен други елемента пара. Захваљујући том механизму, додавање елемента у структуру и претрага захтевају константну временску сложеност, што хеш мапу чини идеалним кандидатом ако је један од захтева извршења временска ефикасност.

Одабир хеш мапе као структуре за чување везе између имена и имплементације функције је имплицирао да веза буде представљена паром [име, имплементација]. Име функције је представљало низ карактера и једноставно је могло да се представи типом *string*. Наредни корак је био дефиниција тога како сама функција треба да буде представљена. У ту сврху искоришћен је механизам делегата

(енглески *delegate*) - конструкт језика *C#* у коме је *QLab* писан. Делегат је тип који представља референцу на методу која прима аргументе одређеног типа и, такође, враћа вредност одређеног типа. Када се креира објекат типа делегат, може да му се додели било која метода која има компатибилан потпис. Делегати се користе како би нпр. методи могла да се проследи метода као аргумент или како би постојало динамичко одлучивање уколико се јави потреба да се позива различита метода, у зависности од тренутне ситуације у којој се извршавање налази. Више о делегатима се може прочитати у [3].

На основу претходно дефинисаног прототипа, дефинисан је делегат који представља функцију:

```
public delegate List<IQLabValue> f(List<IQLabValue> argList);
```

У класи *Functions* у којој су дефинисане функције, дефинисана је и структура која повезује имена функција и имплементације:

```
public static Dictionary<Identifier, f> FunctionsDictionary;
```

и креирана је статичка метода која би требало да буде позвана у току покретања окружења *QLab*, а чија је функција повезивање имена и имплементације:

```
public void addFunctions()
{
      FunctionsDictionary = new Dictionary<Identifier, f>();

      FunctionsDictionary.Add(new Identifier("+"), plus_op);
      FunctionsDictionary.Add(new Identifier("-"), minus_op);
      FunctionsDictionary.Add(new Identifier(".*"), times);
      FunctionsDictionary.Add(new Identifier("./"), rdivide);
      FunctionsDictionary.Add(new Identifier(".\\"), ldivide);
      …
}
```

Ово је омогућило да, касније, свака функција може да се позове веома једноставно. Пример позива једне функције изгледа овако:

```
Functions.FunctionsDictionary [Identifier] (ArgumentsList);
```

Притом се као аргументи прослеђују име функције и аргументи над којима функција треба да се изврши.

2.3. Гаранција квалитета функција

Гаранција квалитета функције је још један од захтева који је требало испунити.

Једна од идеја водиља пројекта *Qlab* је да окружење подржи функционалности *MATLAB*-овог окружења и језика. Та замисао условљава да се свака функција понаша исто као и њен пандан у *MATLAB*-у, тј. да свака функција у *QLab*-у за исте улазне вредности врати исти резултат који враћа та функција у *MATLAB*-овој изведби. Одлучено је да се компатибилност функционалности гарантује механизмом *Unit* тестова. *Unit* тестови су техника тестирања софтвера која се користи најраније у процесу развоја од свих техника тестирања. Поступак представља проверу функционалности најмањих атомичних јединица кода кроз серију тестова који проверавају само да ли задати код, за унапред одређене улазне вредности, враћа унапред договорену излазну вредност. Притом се специјална пажња посвећује томе да дијапазон тестова буде широк како би била покривена сва понашања тестиране јединице кода која су важна. Више о *Unit* тестовима се може прочитати у [4].

Предност коришћења *Unit* тестова у потпројекту математичких функција *QLab*-а не види се само у провери компатибилности са *MATLAB*-ом. Чињеница је да функције *MATLAB*-а нису детаљно документоване на интернету. Имајући то у виду, једини начин да се спознају функционалности појединих функција био је да се састави серија тестова и да се, у зависности од враћених резултата, закључи како функција треба да ради. Формалним записом тих тестова и резултата, направљена је спецификација захтева које функција треба да задовољи.

Такође, написани *Unit* тестови дозвољавају олакшану промену дела кода уколико је потребно да се тако нешто уради у будућности. Уколико се тестови покрену одмах након промене кода, резултати тестова ће показати ако је нека од функционалности изостављена услед превида.

На крају, још једна од предности је и рано откривање проблема и, услед тога, и мала цена решавања тих проблема.

Пример тестова једне функције биће дат касније у делу у којем је објашњен пун поступак израде једне функције.

2.4. Поступак израде функција

У наставку рада је објашњен поступак израде једне функције. Као показни пример изабрана је функција сабирања *plus*, имајући у виду њену једноставну структуру и функционалност.

2.4.1. Фаза прикупљања захтева

Поступак имплементације функције креће прикупљањем захтева. Како детаљна документација није доступна јавности, најбољи начин да се то уради је био да се смисли низ улазних вредности и да се види како се функција у *MATLAB*-у понаша када је позвана са тим улазним вредностима. Притом су вредности морале да буду пажљиво изабране, како би дале одговоре на све недоумице и омогућиле формирање потпуне слике о захтевима.

Основне функционалности сваке функције, које на почетку треба да буду дефинисане, тичу се дозвољених улазних вредности.

Први захтев који је требало да се дефинише је дозвољен број улазних вредности функције. Обично се то установљава позивањем функција са различитим бројем аргумената. Иако постоје функције *MATLAB*-а које могу да имају променљив или неограничен број улазних вредности, *plus* је функција која има само две. У случају мањег или већег броја вредности, исписује се одговарајућа порука.

Други захтев тиче се дозвољених комбинација типова улазних вредности. Такође, овим тестовима се добија и слика везе између типова улазних вредности и типова излазних вредности. Функција *plus* је прилично једноставна и по овом питању. Она за било коју комбинацију нумеричких типова враћа једну вредност типа *float*. Понашање је, такође, исто и за улазне вредности карактерног и логичког типа. Уколико се проследи неки од сложенијих типова о којима је било речи раније, функција исписује поруку да ти типови нису подржани као улазне вредности ове функције.

Следећи корак у фази прикупљања захтева је био да се установи функционалност тестиране функције. Функција *plus* сабира улазне вредности на одређен начин у зависности од изгледа тих вредности. Уколико су улазне вредности истих димензија, излаз је матрица са *float* вредностима тих димензија, где сваки елемент представља збир елемената улазних вредности са одговарајућим координатама. Уколико улазне вредности нису истих димензија, сабирање је могуће само уколико је једна од тих вредности скалар. У тој ситуацији, излаз је матрица типа *float* димензија матрице која није скалар, где сваки елемент представља суму вредности скалара и одговарајуће вредности друге матрице. Уколико су улазне

вредности различитих димензија и ниједна није скалар, приказаће се порука да сабирање оваквих вредности није могуће.

2.4.2. Фаза писања тестова

Након претходно описаних тестова функције *plus* у *MATLAB*-у, они су и формално записани. Ради једноставности, у раду је написан само један тест по захтеву, иако је приликом саме имплементације било потребно написати више од једног теста како би биле покривене све могуће ситуације. Примери тестова изгледају овако:

Тест који проверава како се функција понаша са недозвољеним бројем аргумената:

```
try
{
      QInt64NDimArray element =
            new QInt64NDimArray(
                  new QInt64[] {new QInt64(5)},
                  new int[] { 1, 1 });
      plusArguments.Add(element);

      plusResults = f.plus(plusArguments);

      Assert.AreEqual(true, false);
}
catch (NumericalCalculationsException ex)
{
      Assert.AreEqual(true, true);
}
```

Тест који проверава како се функција понаша кад аргументи нису истих димензија и ниједан од њих није скалар:

```
try
{
      QFloat64NDimArray element1 =
            new QFloat64NDimArray(
                  new QFloat64[]{
                        new QFloat64(double.PositiveInfinity),
                        new QFloat64(5,6)},
                  new int[]{ 1, 2 });
      plusArguments.Add(element1);
```

```
        QFloat64NDimArray element2 =
              new QFloat64NDimArray(
                    new QFloat64[]{
                          new QFloat64(0),
                          new QFloat64(0)},
                    new int[] { 2, 1 });
        plusArguments.Add(element2);

        plusResults = f.plus(plusArguments);

        Assert.AreEqual(true, false);
    }
    catch (NumericalCalculationsException ex)
    {
        Assert.AreEqual(true, true);
    }
```

Тест који проверава како се функција понаша кад су улазне вредности регуларне:

```
    try
    {
        QFloat64NDimArray element1 =
              new QFloat64NDimArray(
                    new QFloat64[]{
                          new QFloat64(double.PositiveInfinity),
                          new QFloat64(5,6)},
                    new int[] { 2, 1 });
        plusArguments.Add(element1);

        QFloat64NDimArray element2 =
              new QFloat64NDimArray(
                    new QFloat64[]{
                          new QFloat64(1),
                          new QFloat64(1)},
                    new int[] { 2, 1 });
        plusArguments.Add(element2);

        QFloat64NDimArray result =
              new QFloat64NDimArray(
                    new QFloat64[]{
                          new QFloat64(double.PositiveInfinity),
                          new QFloat64(6,6)},
                    new int[] { 2, 1 });
        plusResults.Add(result);

        Assert.AreEqual(true,
```

```
            Helper.areIQlabValueListsEqual(plusResults,
                  f.plus(plusArguments));
}
catch (NumericalCalculationsException ex)
{
      Assert.AreEqual(true, false);
}
```

Тест који проверава сабирање аргумената раличитих типова података:

```
try
{
      QCharNDimArray element1 =
            new QCharNDimArray(
                  new QChar[]{new QChar('a')},
                  new int[] { 1, 1 });
      plusArguments.Add(element1);

      QInt32NDimArray element2 =
            new QInt32NDimArray(
                  new QInt32[]{new QInt32(5)},
                  new int[] { 1, 1 });
      plusArguments.Add(element2);

      QFloat64NDimArray result =
            new QFloat64NDimArray(
                  new QFloat64[]{new QFloat64(69)},
                  new int[] { 1, 1 });
      plusResults.Add(result);

      Assert.AreEqual(true,
            Helper.areIQlabValueListsEqual(plusResults,
                  f.plus(plusArguments));
}
catch (NumericalCalculationsException ex)
{
      Assert.AreEqual(true, false);
}
```

Тест који проверава функционалност када су елементи прослеђених аргумената специјалне вредности (бесконачности и *NaN* вредности):

```
try
{
      QFloat64NDimArray element1 =
            new QFloat64NDimArray(
                  new QFloat64[]{
```

```
                        new QFloat64(double.NaN),
                        new QFloat64(double.PositiveInfinity,6),
                        new QFloat64(1),
                        new QFloat64(double.NaN)},
                  new int[] { 2, 2 });
      plusArguments.Add(element1);

      QFloat64NDimArray element2 =
            new QFloat64NDimArray(
                  new QFloat64[]{
                        new QFloat64(10),
                        new QFloat64(15,
            double.NegativeInfinity),
                        new QFloat64(11, 10),
                        new QFloat64(double.PositiveInfinity)},
                  new int[] { 2, 2 });
      plusArguments.Add(element1);

      QFloat64NDimArray result =
            new QFloat64NDimArray(
                  new QFloat64[]{
                        new QFloat64(double.NaN),
                        new QFloat64(double.PositiveInfinity, dou
            ble.NegativeInfinity),
                        new QFloat64(12, 10),
                        new QFloat64(double.NaN)},
                  new int[] { 2, 2 });
      plusResults.Add(result);

      Assert.AreEqual(true,
            Helper.areIQlabValueListsEqual(plusResults,
                  f.plus(plusArguments));
}
catch (NumericalCalculationsException ex)
{
      Assert.AreEqual(true, false);
}
```

2.4.3. Фаза писања функције

Пошто су претходне фазе развоја функције завршене, сви захтеви познати и записани, кренуло се са имплементацијом саме функције. Имајући у виду претходно дефинисане захтеве, тело функције *plus* након израде изгледа овако:

```
public List<IQLabValue> plus(List<IQLabValue> argumentList)
{
      // Check if argument list has 2 elements
      if (argumentList.Count > 2)
            throw (new NumericalCalculationsException(
                  "??? Error using ==> plus \n Too many input arguments
            ."));

      if (argumentList.Count < 2)
            throw (new NumericalCalculationsException(
                  "??? Error using ==> plus \n Not enough input argumen
            ts."));

       // Try to cast to float64 (that's how MATLAB works);
      //  if it does not work, it is CELL, STRUCT or something invalid
      if (Helper.canConvertListToEnum(argumentList,
                              IQLabValueType.QFloat64NDimArray))
      {
            // Convert to float
            List<QFloat64NDimArray> floatList
                  = new List<QFloat64NDimArray>();
            foreach (var element in argumentList)
                  floatList.Add(new QFloat64NDimArray(element));

            // Check if one of them is scalar (1x1 matrix)
            bool firstIsScalar =
                  Helper.checkIfScalar (floatList.ElementAt (0));
            bool secondIsScalar =
                  Helper.checkIfScalar (floatList.ElementAt (1));
            if (firstIsScalar || secondIsScalar)
            {
                  QFloat64NDimArray scalar;
                  QFloat64NDimArray otherOne;
                  if (firstIsScalar)
                  {
                        scalar = floatList.ElementAt (0);
                        otherOne = floatList.ElementAt (1);
                  }
                  else
                  {
                        scalar = floatList.ElementAt (1);
                        otherOne = floatList.ElementAt (0);
                  }

                  // Get scalar value
                  QFloat64 scalarValue = scalar.First;

                  // Create the result
```

```
      // It is the same dimension as the other matrix
      QFloat64NDimArray result =
            new QFloat64NDimArray (otherOne.DimensionSize);

      // Iterate through another matrix and
      // Add scalar to every element
      Counter c =
            new Counter (otherOne.DimensionSize);
      for (c.reset; c.plusPlus();!c.isAtEnd())
      {
            // Get element of the other matrix
            QFloat64 element = otherOne [c.getCurrent()];

            // Do the sum
            QFloat64 resultElement =
                  new QFloat64NDimArray();
            resultElement.Real = element.Real
                                    + scalarValue.Real;
            resultElement.Imag = element.Imag
                                    + scalarValue.Imag;
            result [c.getCurrent()] = resultElement;
      }

      // Create result list and return it
      List<IQLabValue> resultList = new List<IQLabValue>();
      resultList.Add(result);
      return resultList;
}

// At his point, we are sure they are both not scalars
if (Helper.checkDimensions(
      floatList.ElementAt(0),
            floatList.ElementAt(1)) == false)
      throw( new NumericalCalculationsException(
            "Error using ==>
                  plus \n Matrix dimensions must agree."));
else
// The dimensions match
{
      QFloat64NDimArray first = floatList.ElementAt(0);
      QFloat64NDimArray second = floatList.ElementAt(1);
      int[] dimensions = first.DimensionSize;

      // Create result
      QFloat64NDimArray result =
            new QFloat64NDimArray(dimensions);
      // Iterate through matrix
      Counter c= new Counter(dimensions);
```

```
                for (c.reset(); c.plusPlus (); !c.isAtEnd ())
                {

                      // Get proper element
                      QFloat64 firstElement =
                            first [c.getCurrent ()];
                      QFloat64 secondElement =
                            second [c.getCurrent ()];

                                  // Do the sum
                      QFloat64 resultElement =
                            new QFloat64NDimArray ();
                      resultElement.Real =
                            firstElement.Real + secondElement.Real;
                      resultElement.Imag =
                            firstElement.Imag + secondElement.Imag;

                      // Set the result element
                      result [c.getCurrent ()] = resultElement;
                }

                // Create result list and return it
                List<IQLabValue> resultList =
                      new List<IQLabValue> ();
                resultList.Add (result);
                return resultList;
          }
     }
     else
     // Converting failed, these argument are invalid
     {
          throw (new NumericalCalculationsException(
                "??? Undefined function or method 'plus'
                      for input arguments of type  “+
                            Helper.PrintEnumName(
                                  Helper.TypeOfThatCouldNotConvert(
                                        argumentList,,
                                  IQLabValueType.QFloat64NDimArray))
                                        + "."));
     }
}
```

Као што је раније описано, функција враћа резултат као први елемент листе која представља излазну вредност. У одређеним ситуацијама, потребно је да се функција не изврши и да се у том тренутку прикаже порука кориснику. То се најчешће дешава кад улазне вредности нису подржане. Ова функционалност

изведена је коришћењем механизма изузетака. Део кода који је позвао функцију може да прочита поруку која је намењена кориснику током обраде изузетка.

2.5. Урађене функције

Претходно описани поступак је поновљен за сваку од имплементираних функција.

У Табели 1. су пописане све урађене функције груписане на основу сличних функционалности.

Табела 1. Листа урађених функција

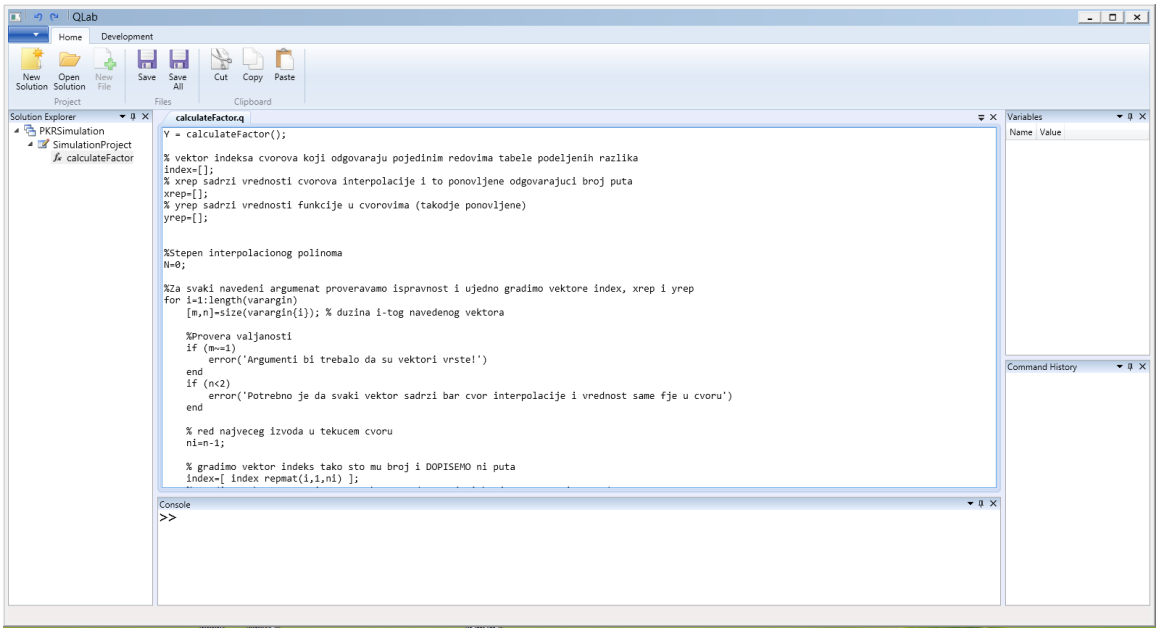
Функције	Објашњење
plus, minus, uplus, uminus, times, abs, min, max, nzmax, exp, reallog, log, log2, mod, rem, ndims, nnz, pow2, power, realpower, sqrt, realsqrt, sign, hypot, sum, prod, cumsum, cumprod	Функције ове групе представљају неке од основних математичких операција, које се често користе.
rdivide, ldivide, ctranspose, transpose, mtimes	Група функција која представља основне операције са дводимензионалним матрицама.
ge, gt, le, lt, eq, ne, isequal, isequalwithequalnans	Функције koje се користе за поређење вредности.
and, not, or, xor	Основне функције за манипулацију логичким вредностима.
cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, acosh, asinh, atanh, atan2	Група тригонометријских функција.
bitmax, bitcmp, bitget, bitset, bitshift, bitand, bitor, bitxor	Група битовских операција.
ceil, floor, round, fix	Функције које се користе за заокруживање вредности са покретним зарезом.
complex, conj, real, imag, false, true, zeros, nonzeros, ones, eye, all, any, inf, nan, triu, tril, diag, rand, randi, randn	Група функција које се користе за генерисање нових низова на различите начине.
logical, char, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, double	Функције ове групе користе се за превођење вредности једног типа у вредност другог типа.
ischar, isempty, isfinite, isnan, isinf, isfloat, isinteger, isletter, isnumeric,	Функције које се користе за испитивање различитих својстава

isreal, isscalar, isspace, isvector, iscell, isvarname, isstruct, isa, isstr, isstrprop, issorted	неких вредности.
deblank, findstr, lower, upper, strrep, strcmp, strcmpi, strncmpi, length, strncmp, regexp, regexpi	Група функција за манипулацију стринговима.
size, sort, permute, find, reshape, filter, colon	Функције које служе за манипулацију низовима.
namelengthmax, numel, getenv, setenv	Функције које се користе за подешавање окружења математичких функција.

3. Закључак

Пројекти отвореног кода могу да буду од великог значаја за академску заједницу. Овакви пројекти доприносе угледу факултета и пружају шансу да се факултет повеже са другим институцијама које се баве сличним областима, чиме се отварају нове могућности за будућност.

Током досадашњег рада на потпројекту математичких функција дефинисане су и написане структуре података које служе за представљање типова података у *MATLAB*-у, направљен је интерфејс који омогућава једноставно позивање функција из других делова кода, направљен је механизам који обезбеђује квалитет функција и имплементиране су функције које нису биле зависне од других потпројеката. Истовремено, направљен је и значајан помак на *Parser*, *Engine* и *GUI* потпројектима. То је омогућило да на интернету постане доступна прва (алфа) верзија окружења *QLab*, која је у стању да изврши основна израчунавања написана у језику *MATLAB*. На Слици 3. је приказан изглед тренутне верзије апликације.



Слика 3. Изглед апликације *QLab*

Гледајући урађено, чини се да је утицај који је *QLab* имао на студенте јако велики. Стварање овог пројекта је за многе студенте представљало шансу да стекну преко потребну праксу и искуство. *QLab* је студентима омогућио први сусрет са организацијом пројекта, постављањем циљева, израдом архитектуре система, коришћењем механизама за верзионисање кода и тестирањем софтвера. Пружајући то искуство, *QLab* је постао јако важна одскочна даска у каријерама тих

студената који сада раде у водећим софтверским компанијама у нашој земљи. Тренутно, *QLab* је још у фази развоја и шанса да нови студенти Математичког факултета добију преко потребно искуство је и даље ту. Та шанса ће несумњиво бити искоришћена и број младих људи на чији је живот овај пројекат утицао позитивно ће само расти са временом.

4. Литература

[1] Sharma, Neeraj; Gobbert, Matthias K. – A COMPARATIVE EVALUATION OF MATLAB, OCTAVE, FREEMAT AND SCILAB FOR RESEARCH AND TEACHING

[2] Детаљи о *CSV* формату за чување података: https://en.wikipedia.org/wiki/Comma-separated_values

[3] Albahary, Joseph; Alabahary, Ben - C# 5.0 in a Nutshell: The Definitive Reference

[4] Osherove, Roy - The Art of Unit Testing: With Examples in .Net