Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Interpoláció osztott rendszereken

Tejfel Máté
egyetemi tanár

Cselyuszka Alexandra
Informatika Bsc

Budapest, 2015

Tartalomjegyzék

1. fejezet

Bevezetés

"A gyakorlatban sokszor felmerül olyan probléma, hogy egy nagyon költségesen kiszámítható függvénnyel kellene egy megadott intervallumon dolgoznunk. Ekkor például azt tehetjük, hogy néhány pontban kiszámítjuk a függvény értékét, majd keresünk olyan egyszerűbben számítható függvényt, amelyik illeszkedik az adott pontokra." [1]

A szakdolgozatom célja ezekre a problémákra megoldást adni elosztott környezetben.

1.1. Feladat elemzése

Adott ponthalmazokból kívánunk egy közelítő polinomot becsülni. Ezeket különböző interpolációs technikával meg tudjuk adni, ki tudjuk számolni. Több interpolációs technika létezik, melyekből könnyen meg tudunk adni akár több polinomot is egy adott ponthalmazhoz.
Ezekkel a számításokkal előfordulhat, hogy lassan futnak, főleg ha több interpolációt kívánunk egyszerre számolni. Ebben az esetben optimálisabb több gépen számolni a különböző ponthalmazokat.

Ebben a feladatban egy speciális megvalósítása lesz ennek a számításnak.
A megvalósítás grafikus része egy weboldal, melyen szerkeszthetjük a ponthalmazokat. A számítás részét egy szerver végzi amely figyeli a felcsatlakozó gépeket. Amikor kap egy számítandó adathalmazt, akkor több gép segítségével kiszámítja az eredményt. Ha minden részfeladat végzett, akkor visszaküldi a weboldalra, ahol az eredmények megtekinthetőek grafikus formában.

1.2. Feladat megvalósítása

A **grafikus felület** egy weboldal, mely JavaScript-ben és HTML-ben készült. A felületen egy listát tekinthetünk meg, ahova több ponthalmazt is felvehetünk.
Mentés hatására az értékek a háttérben eltárolódnak. A ponthalmazok közül választhatunk egyet, amely betöltődik a felületre.
A szerkesztő felület egy táblázatból és egy grafikonból áll, emellett még a különböző speciális számításra vonatkozó tulajdonságok (interpoláció típusa) valamint a grafikonon való megjelenítéshez tartozó tulajdonságok (polinom pontosság, megtekintendő intervallum) is szerkeszthetőek.
Ha befejeztük a halmazok szerkesztését elküldhetjük a számítani kívánt értékeket a szerver felé.

A **szerver** feladata hogy figyelje a felületről érkező adatokat. Ha az adathalmaz megérkezett, akkor a szerver kibontja az adatokat egy JSON-ból, és elindítja az elosztást.
Az elosztáshoz a szerveren el kell indítani egy figyelő folyamatot amelyre lehetősége van egy külső gépnek felcsatlakozni. Amikor a szerveren indul egy számolás a felcsatlakozott gépeket lekérdezi, majd a feladatokat szétosztja.
A szerver megvalósítása és a gépekre való szétosztás Erlang-ban lett megvalósítva. A JSON feldolgozásához mochi-json lett alkalmazva. A feldolgozás után az adathalmazon végig megyünk és azok alapján felparaméterezzük, és meghívjuk a számítást végző függvényt.
A számításhoz használt maximális gépek száma paraméterként megadható, de a tényleges számítást csak annyi gépen tudjuk maximálisan végezni ahány gép felcsatlakozott a számításhoz.

A **számítás** megvalósítása C++ nyelven történt. A paraméterek alapján a Lagrange -féle, Newton -féle, Hermite -féle interpolációs technikák közül eldönti melyik esetet használja.
A programban kellett implemetálni egy egyszerű polinom szorzás és összeadást, valamint az interpolációkhoz szükséges függvényeket. A Lagrange számítás a polinom műveletek és a képlet felhasználásával ciklusokkal valósul meg. Newton és Hermite esetén a kapott adatokból először a kezdő mátrixot kell legenerálni, majd kiszámítani.
Abban az esetben ha Newton vagy Lagrange polinomot számolunk nem vesszük figyelembe a derivált pontokat, viszont figyelembe vesszük ha inverz számítást kívánunk végezni.

2. fejezet

Felhasználói dokumentáció

2.1. Bevezetés

2.2. Telepítési útmutató

2.2.1. Rendszer követelmények

2.2.2. Segédprogramok telepítése

2.2.3. Szerver és segédgépek üzembe helyezése

2.2.4. Használati útmutató

Weboldal

Szerver

3. fejezet

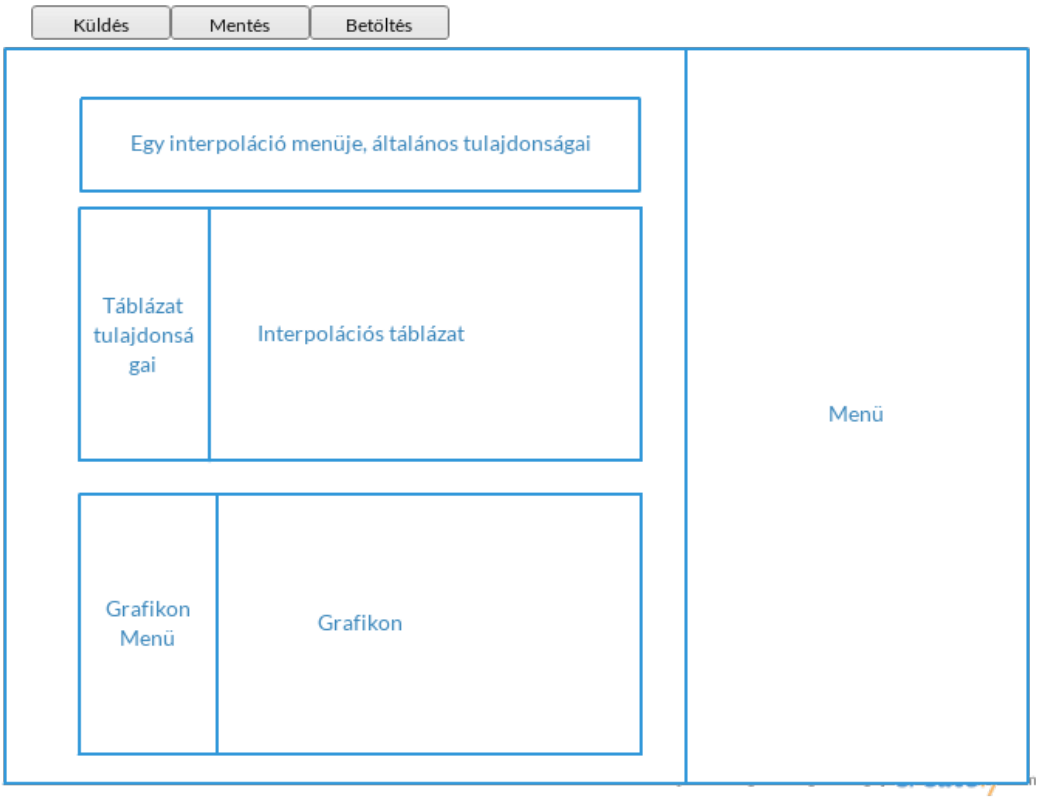
Fejlesztői dokumentáció

3.1. Megoldási terv

A program működése 2 részre bontható: weboldalra(kliens) és a szerverre. A weboldalon össze állított adatokat küldjük fel a szerverre, a szerver a megkapott adatok alapján számol.

3.1.1. Weboldal

A kliens megvalósításához az alábbi technológiák merültek fel : C++/Qt, C#, JS/HTML. Végül JavaScript-ben lett megvalósítva, első sorban a grafikon kirajzoló (Flot) miatt, de a szerver kommunikáció egyszerűsége is döntő ok volt amellett, hogy egy weboldal bármely gépen egyszerűen megnyitható, kezelhető.
Egy oldalból áll melyen a felhasználó szerkesztheti az adatokat. Azért nem lettek a részek külön oldalakon megvalósítva, mert az egyik oldalról az adatok átvitele egy másik oldalra nem annyira egyszerű, viszont nincs is olyan komplex az oldal, hogy szükséges legyen több aloldalra szétbontani. Az oldal megjelenés felépítése megtekinthető 3.1-es képen. A weboldalon meg kell valósítani a pontok dinamikus kirajzolását, és a táblázatos formában történő megjelenítést és szerkeszthetőséget. Mivel több interpolációt küldünk fel a szervernek ezért a weboldalon több szerkesztésére is lehetőséget kell adni.
Több interpoláció számítás szerkesztésének megvalósításához kell egy menü rendszer, amelyben eltárolódnak az adatok, és képesek betöltődni.
Az oldalon input-okat használunk még, és a dinamikus táblázat is JavaScript-ből van legenerálva. A táblázatokban sorok beszúrására teljes táblázat törlésre is lehetőséget kell adni. A táblázatokban inputok vannak az egyes cellákban melyben egyszerű értékek, vagy akár komplexebb Objektumok is találhatóak. A bonyolultabb objektumokat json string-ben tároljuk ezekben az inputokban.



3.1. ábra. Weboldal vázlata

A menü listájában új adathalmazokat hozhat létre, a régieket szerkesztheti. Amikor a felhasználó pontokat, megjelenítést frissít a legtöbb esetben az oldal már a háttérben menti az adatokat a listába. Amikor egy másik interpolációt választunk ki, akkor az betöltődik a táblázatba, és a grafikonba. A módosítások és mentés esetén az aktuálisan kiválasztott interpolációs adathalmaz sora fog frissülni. Ha a felhasználó végzett egy gombra nyomással a program legenerálja a szükséges objektumot.
A felület sok gombot tartalmaz, melyek hatására frissíthetőek az adatok. Amikor frissítünk egy részt, általában mentődnek az értékek egy inputba JSON formában.

3.1.2. Elosztott rendszer

Az elosztott rendszer megvalósításához az alábbi technológiák merültek fel: C++/PVM, Erlang. Miután a JavaScript mellett döntöttem a grafikus felületen, ezután optimálisabbnak tűnt egy hasonlóan gyengén típusos nyelvnek a használata. Az Erlang elég jól támogatja párhuzamosítást és a szerver kommunikációt is, és bár az algoritmusok implementálása nehézkesebb lett volna, de C++-ban megvalósított függvények beépítése miatt ez a probléma megoldódott.
TODO : "nodeHandler"

3.1.3. Számítás

A számítás megvalósításánál felmerült hogy Erlang-ban legyen, de mivel a számítást ciklusokkal érdemes megvalósítani, ezért egyszerűbb volt egy nem funkcionális nyelvben implementálni azokat. A C++-os függvényeket fel lehetett használni az Erlang modulokban.

3.1.4. Kommunikáció

Amikor a szervert létrehozzuk akkor inicializálunk két processzt. Az egyik a kliens felől várakozik kérésre, a másik a node-ok felől. Amikor egy node fel kíván csatlakozni küld egy kérést. Ha sikeres volt akkor a szerver ezt jelzi neki, és felkerül a listára. A weboldalon egy gomb hatására megy egy kérés a szerver felé. A szerver jó esetben fogadja a kérést. Ha nem sikerült kapcsolatba lépnie a szerverrel, akkor jelzi a felhasználónak hogy a kapcsolódás során hiba lépett fel.
Ha fogadta a kérést, akkor megpróbálja feldolgozni az adatokat. Ha sikeresen feldolgozta az adatokat, abban az esetben elindul a szétosztás.
A szétosztás során a felcsatlakozott gépeken létre jönnek a processzek, majd kapnak egy adathalmazt mellyel számolniuk kell. Ha végeztek, az eredményt visszaküldik a szülő processznek. A szülő processz, ha megkapott minden értéket, azt visszaküldi a weboldalnak.
A weboldal sikeres válasz után betölti az eredményeket.

3.2. Felhasznált források és alkalmazásuk

3.2.1. Grafikon kirajzoló - Flot

A Grafikon megjelenítéséhez a Flot-ot használom. Ez egy jQuery-s könyvtár, melyben egyszerűen és látványosan lehet grafikonokat kirajzolni. A forrás a "WebPage/source/flot-8.0.2" mappában érhető el.

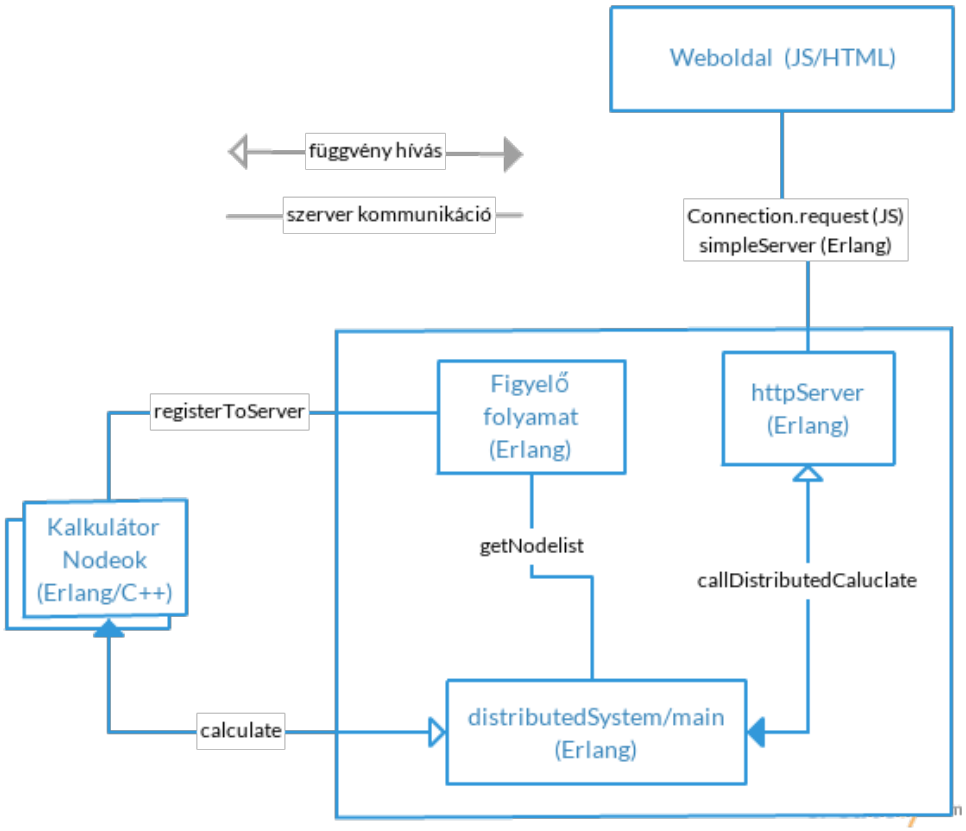
Felhasználás

HTML fájlban egy egyszerű "DIV"-ként jelenik meg, melyet aztán a JavaScript tölt meg tartalommal.

```
<div id=\"resultplot\" class=\"demo-placeholder\"></div>
```

A JavaScript-ben hivatkozhatunk erre a "DIV"-re, majd az adatok és a típusok segítségével alábbi módon hivatkozhatjuk meg:

```
var placeholder = $("#resultplot");
var plot = $.plot(placeholder, flot_data, type);
```



3.2. ábra. Kommunikáció

A paraméterezése a grafikon kirajzolónak az alábbi:

placeholder

DIV hivatkozása

type

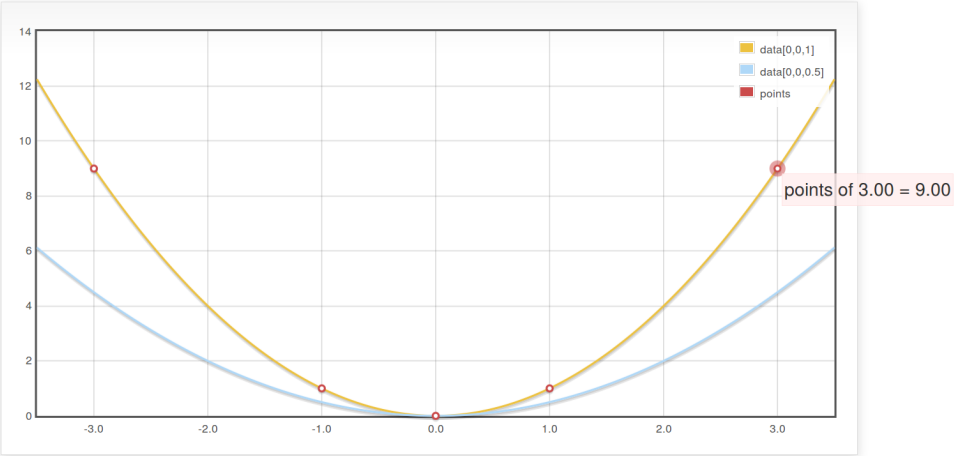
Megjelenítendő grafikon típusa

A Flot sok lehetőséget nyújt a típusok kiválasztására, és ezekre való példákból megalkottam a saját típusomat mely a következőket tartalmazza egy objektumban:

```
series: { line: { show: true } }
```

Beállítjuk hogy a vonalakat jelenítse meg. Ekkor a pontokat is megjeleníti, a többi beállítás függvényében.

```
xaxis: { zoomRange: [0.1, 1], panRange: [-1000, 1000] }
yaxis: { zoomRange: [0.1, 100], panRange: [-1000, 1000] }
```



3.3. ábra. Grafikon kirajzoló

X és Y koordinátákon nagyítás és mozgatási beállítások interaktívvá állítása

```
grid: { hoverable: true, clickable: true }
```

Ezeket a tulajdonságokat használjuk arra hogy felvegyünk új pontokat.
Emellett ha ráviszem az egeret az egyik pontra, megmutatja a pont koordinátáját, és értékeit, és hogy melyik ponthalmazon van.

```
zoom: { interactive: true}, pan: { interactive: true }
```

Nagyítás és kattintással mozgatás engedélyezése.
Ennek a beépítésével is foglalkoztam, de a kattintás sajnos nem egyeztethető könnyen össze a pont figyeléssel, valamint a beépítés után lassú lett, és akadozott a felület, így végül az interaktivitását külső komponensekkel(inputokkal) oldottam meg.

flot_data

A tényleges adathalmazokat tartalmazó tömb, melyben az egyes adatokról egyéni információkat is tartalmazza.

data

Pontok halmaza, melyeket megjelenítünk
[x, y] pontokból álló tömb
Polinom esetén is ezt használjuk, ezért a polinom behelyettesített értékeit adjuk itt meg. Amikor az egérrel felé megyünk ezeket a pontokat fogja megjeleníteni.

label

Adathalmaz elnevezése, ezt láthatjuk amikor az egérrel a pont felé visszük az egeret, valamint a színek-elnevezések össze párosításánál is segít.

points

Ha pontokat kívánunk megjeleníteni, akkor ezt a kapcsolót kell alkalmazni.

lines

Ha a pontokból alkotott vonalat kívánunk látni, akkor ezt a kapcsolót kell alkalmazni. Ezt használjuk a polinom megjelenítéséhez.

```
var example_datas = [{
    data: d4,
    label: "neved4",
    lines: { show: true }
}, {
    data: d3,
    label: "neved43"
    points: { show: true }
}];
```

3.2.2. Http szerver

A szerver kommunikációt Erlang oldalon 2 megtalált minta fájlból állítottam elő. Az egyikben egy egyszerű szerver kommunikációt mutattak be Erlang-ban[3].
A másik pedig az Erlang dokumentációjában megtalálható minta kommunikáció tcp protokollal[4].

3.2.3. Ping-pong Node figyelő

http://www.erlang.org/doc/getting_started/conc_prog.html Az Erlang oldalán található dokumentációban, mely az elosztás és a node kommunikációt mutatja be, sok minta kódot tartalmaz, melyekből könnyen elő tudtam állítani a saját kódomat.
Ez alapján a párhuzamosítottan számoló programomat könnyen át tudtam alakítani elosztott módon számítóvá.
Emelett kellett valami lehetőség arra hogy a gépek fel tudjanak csatlakozni a szerverre. Lehetőség lett volna arra is hogy a figyelő helyett csak megkapja a gépek listáját a szerver, de így tisztábban szét van választva a háttér és a kliens, és nem is szükségesek a tényleges számításhoz.
Ezen az oldalon található minták a ping-pong kommunikációra, melynek segítségével hoztam létre a node-figyelőt. Ez a modul a `ServerConfig/nodeWatcher.erl`

fájlban található meg.

Ennek mintájára hoztam létre az alábbi figyelőt, melyet regisztrálunk `pid_watcher` atom segítségével.

```
startPidWatch() ->
    PidWatch = spawn(nodeWatcher, pidWatch, [self(), []]),
    register(pid_watcher, PidWatch),
    PidWatch.
```

A másik gépről ezután lehet küldeni egy "ping"-et, melyet megkap a node-figyelő.

```
registerToServer(Pong_Node) ->
    spawn(nodeWatcher, registerToServerNode, [Pong_Node]).
...
registerToServerNode(Pong_Node) ->
    {pid_watcher, Pong_Node} ! {worker_write, self(), node()},
    ...
```

3.2.4. Struktúra kezelő: mochijson

A mochijson egy Erlang-hoz is használt modul, melynek segítségével egy json string-et át lehet konvertálni Erlang-os struktúrává.
`git clone https://github.com/mochi/mochiweb.git`(2015.05) paranccsal a teljes mochiweb könyvtárat le lehet tölteni. A `mochiweb/src` mappában találhatóak azok a fájlok melyeket le lehet fordítani, és be lehet tölteni az Erlang shell-be. Lehet letölteni a fájlokat különállóan is.
Eredetileg csak a mochijson.erl-re volt szükségem, de miután komplexebb adatot kellett encode-olnom szükségessé vált még egy fájl betöltése, viszont nem tudtam hogy az még milyen függőségeket hordoz magában, így letöltöttem az egész repository-t. Ha Erlang-ban lefordítjuk shell-ben `mochijson:encode, decode` függvényekkel egyszerűen lehet használni.
Példa képen megtekinthetjük az alábbi egyszerű json-t, mely már tartalmaz objektumot és tömböt.

```
{
    "1": {
        "result": [
            0.1,
            0.5,
            0.7
        ],
```

```
        "time": 0.5
    }
}
```

Ennek string formájára meghívhatjuk a mochijson decode függvényt.

```
ErlStruct = mochijson:decode(
    "{\"1\":{\"result\":[0.1,0.5,0.7],\"time\":0.5}}"
).
```

Ennek eredménye egy Erlang-os struktúra.

```
{struct, [
    {"1", {struct,[
          {"result",{array,[0.1,0.5,0.7]}},
          {"time",0.5}
    ]}}
]}
```

Az `mochijson:encode` segítségével pedig ezt a struktúrát vissza tujduk alakítani json string-gé. Bár az eredményt shell-ben binary formában lehet egyszerűen megtekinteni, és ebben a formában is kell vissza küldeni a kliensnek.

```
iolist_to_binary(mochijson:encode(ErlStruct)).
<<"{\"1\":{\"result\":[0.1,0.5,0.7],\"time\":0.5}}">>
```

Felhasználás

Ennek a típusa nem egyszerű, főleg ilyen komplex adathalmaznál. Ennek kezelésére létre lett hozva a Utility/structHandler modul.
Első lépésben a string-et konvertáljuk a kellő típussá.

```
getDataByJson(JsonSting) -> apply(mochijson, decode, [JsonSting]).
```

Ismeretek segítségével létre hoztam egy olyan függvényt, ami kinyer egy adott értéket a struktúrából.

```
getElementByKey(array, {array, Array}) -> Array;
getElementByKey(struct, {struct, Array}) -> Array;
getElementByKey(Name, {struct, Struct}) -> getElementByKey(Name, Struct);
getElementByKey(Name,[{Name, Value}]) -> Value;
```

Ennek felhasználásával dolgoztam fel a struktúrát.

3.2.5. Erlang modul C++-ban : NIF

NIF [2] (native implemented functions) egy C könyvtár, melynek segítségével implementálhatjuk egy modul függvényeit C-ben vagy C++-ban.

Felhasználás

A calculator Erlang-modul megvalósítását C++-ban implementáltuk.
Calculator/ mappában található erlang.cpp és a calculator.erl fájlokban használjuk fel a NIF-et. Amely fájlban végezzük a modul közvetlen kommunikációját az Erlang-al ott le kell tölteni az `erl_nif` könyvtárat.

```
#include "erl_nif.h"
```

Emellett meg kell neki adni, melyik függvényeket, hány paraméterrel szeretnénk az Erlang-ba betölteni. A fájlt `Calculator/erlang.cpp` néven találhatjuk meg.

```
static ErlNifFunc nif_funcs[] = {
    {"calculate", 4, calculate_nif}
};
```

Amikor inicializáljuk, le kell írni hogy melyik modul-nak lesz a része.

```
ERL_NIF_INIT(calculator, nif_funcs, NULL, NULL, NULL, NULL)
```

Amikor meghívódik az Erlang-ból ez a modul, paraméterként `ERL_NIF_TERM` típusú változókat kapunk, melyeket lehetőség van átkonvertálni C++-os típusokká.
Az `enif_get_int` függvény segítségével egy változóból kinyerhetjük az integer-ré konvertált értékét. Hasonlóan használjuk az `enif_get_double` függvényt mely értelem szerűen egy double típusú értéket ad vissza.
Egy lista elemeit `enif_get_list_cell` függvény segítségével tudtam átkonvertálni. Ennek segítségével megvalósítottam a vektor-rá és mátrix-á alakító függvényeket,

```
ERL_NIF_TERM head; ERL_NIF_TERM tail = arg;
//...
while(enif_get_list_cell(env, tail, &head, &tail)) {
    //...
}
```

A számítás eredménye egy vektor, így az eredményül kapott értéket vissza kell valahogyan konvertálni Erlang-os típussá. Ezt az `enif_make_double` és az `enif_make_list_from_arra`
függvények segítségével lehet megvalósítani. Ennek implementálása a `convertList` függvényben történt.
A tényleges híváshoz a `nif_funcs`-ban meghivatkozott `calculate_nif`-et kell implementálni.

Ebben meghívódnak a kovertáló függvények a helyes paraméterezés kialakításához, majd az eredménnyel vissza térünk egy lista formájában.

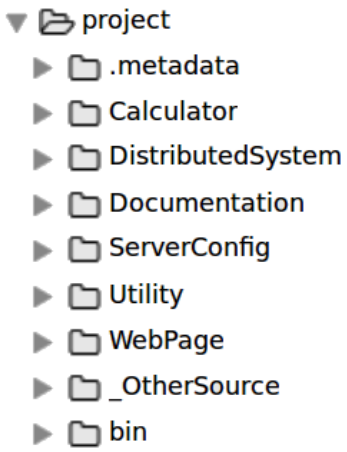
```
static ERL_NIF_TERM calculate_nif(ErlNifEnv* env, int argc, const ERL_NIF_TERM ar
    //...
    poli = interpolateMain(X, Y, type, isInverse);
    return convertList(env, poli);
}
```

3.2.6. Dokumentációhoz felhasznált programok

A tervezési, és megvalósítási szemléltető képek és diagramok egy webes szerkesztő segítségével valósultak meg. Ez az oldal 2015-ben a http://creately.com/ címen volt elérhető.

3.3. Megvalósított mappaszerkezet

A fájlszerkezetnek a tervezésénél nem voltak a belső részek komolyabban megtervezve, dinamikusan változtak. 3 mappa volt a tervezett: elosztott rendszer, weboldal és a külső fájlok mappája. A többi mappa mind a külsők, és a belsők a fejlesztés során lettek bele tervezve, amikor a fájlok mennyisége, vagy az új technológia szeparálása megkívánta azt.



3.4. ábra. Külső mappaszerkezet

Calculator

Az interpoláció számítás megvalósítását valamint az Erlang-modullá alakítását is tartalmazó mappa. Függvényei a DistributedSystem mappából hívódnak.

DistributedSystem
: Elosztás logikáját tartamazó mappa, melynek függvényei a ServerConfig mappából hívódnak.

ServerConfig
: Szerver megvalósítását tartalmazó mappa.

Utility
: Több helyen is meghívható függvényeket tartalmazó mappa. Struktúra kezelés megvalósítása és a mochijson található itt.

WebPage
: Weboldal mappája.

_OtherSource
: Külső felhasznált komponensek mappája.

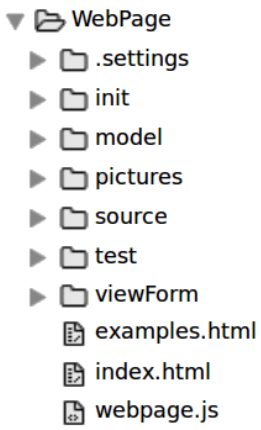
bin
: Szerver inicializálását tartalmazó mappa. Ide kerülnek a lefordított fájlok is.

Documentation
: Dokumentáció forrás fájljait tartalmazó mappa.

3.4. Weboldal megvalósítása

3.4.1. Felépítés

A weboldal forráskódja a /webpage mappában helyezkedik el. A fájlszerkezet az alábbi:



3.5. ábra. A weboldal mappái

webpage.js :
Globális változók inicializálása és pár alapbeállítás lefuttatása

webpage.html :
Weboldal megjelenítése, fájlok betöltése

/init
Inicializáló függvények hívásai és események

menulist.js :
Interpolációk listájának inicializálója

plot.js :
Interpolációs grafikon inicializálása

table.js :
Interpolációs táblázat inicializálása

events.js :
Gombra kattintások eseményei

/model
Objektumok, melyeket az inicializáló lépésben hívunk, és azok segédletei

base.js :
Globális függvények
Base.get, Base.erlangJSON, Base.forEach

base_table.js :
Általános táblázat generáló függvény

connection.js :
Szerver kapcsolat meghívására szolgáló függvény
Connection.request

plot_types.js :
A grafikon kirajzoló típus objektumai

polinome.js :
Polinom kirajzolását segítő függvények
makePolinome található benne és egyéb segédfüggvények

web_page_debug.js :
A Weboldalon történő kiíratást segítő objektum
Jelenleg sehol nem használjuk már, de a megvalósítás során fontos szerepe volt a hibajavításban

/model/interpolation

Az oldal 3 fő részegységének függvényei

menulist.js :

Interpolációk lista megvalósítása
function interpolationMenulist (aConfig) Objektum fájlja

plot.js :

Interpolációs grafikon megvalósítása
function interpolationPlot(aConfig) Objektum fájlja

table.js :

Interpolációs táblázat megvalósítása
function interpolationTable(aConfig) Objektum fájlja

/test

Olyan weboldal részlet fájlok, melyekből kialakult a mostani nagy fájl, és az objektumai, valamint tartalmaz még minta adathalmazokat.

/viewForm

Megjelenítéssel kapcsolatos css fájlokat tartalmazó mappa.

3.4.2. Fontosabb objektumok és függvények

Az objektumokat legtöbb esetben egy függvény generálja, melyeket a visszatérés után felhasználunk az eseménykezelésekhez.

makePolinome(inPolinome, plotFor)

Polinom pontjainak legenerálására szolgáló függvény, a grafikon kirajzolónak megfelelő típusban

inPolinome

A polinom tömbös formában.

plotFor

A polinom intervalluma és pontossága.

Connection.request(aConfig)

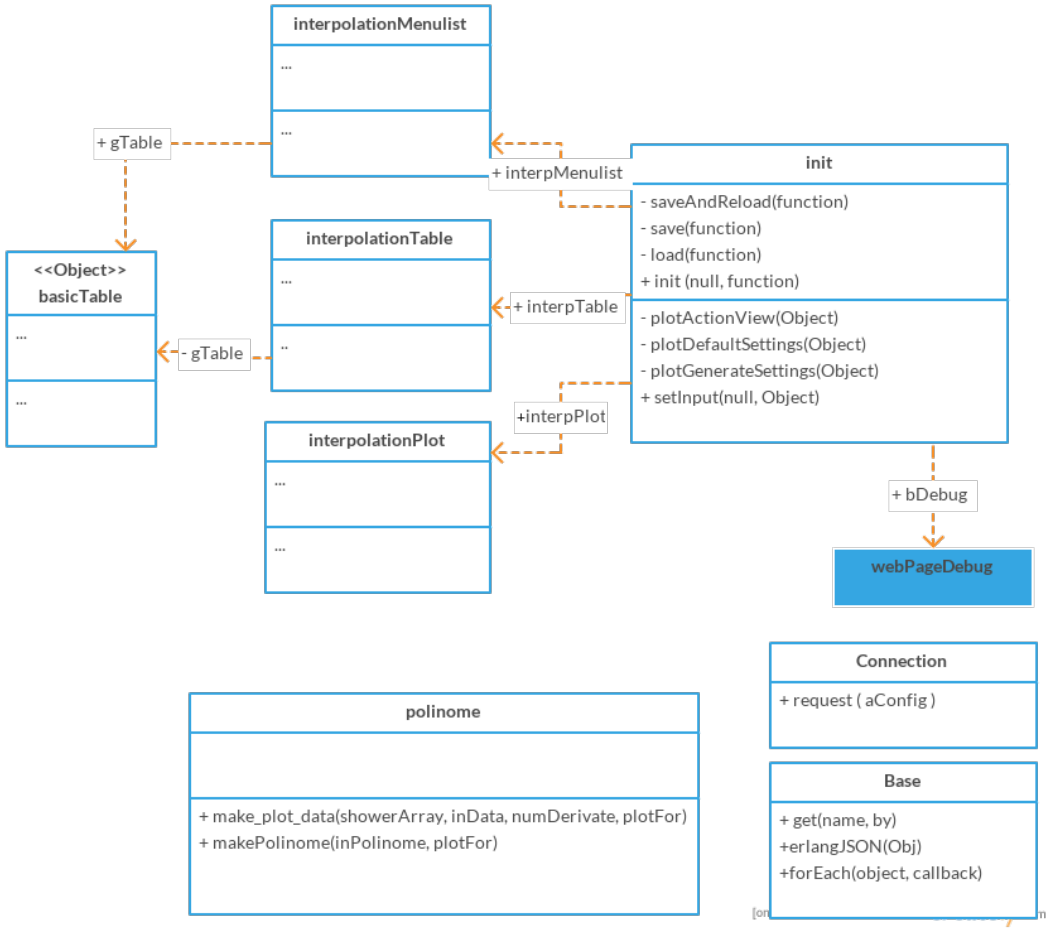
Elküldi a szervernek az értékeket

aConfig.params

A kommunikációban a paraméter amelyet átküldünk a szervernek

aConfig.callback

Sikeres visszatérés esetén ez a függvény fut le a szervertől vissza adott válasszal.



3.6. ábra. Weboldal osztálydiagramja

basicTable (aConfig)

Egy alap tábla objektum. Ennek segítségével lehet létrehozni az interpolációs táblázatot és a menü listát(interpoláció választó).

that.addNewCellToRow(rowIndex, textValue, inputAttributes)

Ad egy új cellát a sorhoz.

that.addNewRowToTable(data)

Ad egy új sort a táblázathoz.

that.addNewColumnToTable(data)

Ad egy új oszlopot a táblázathoz.

that.newTable()

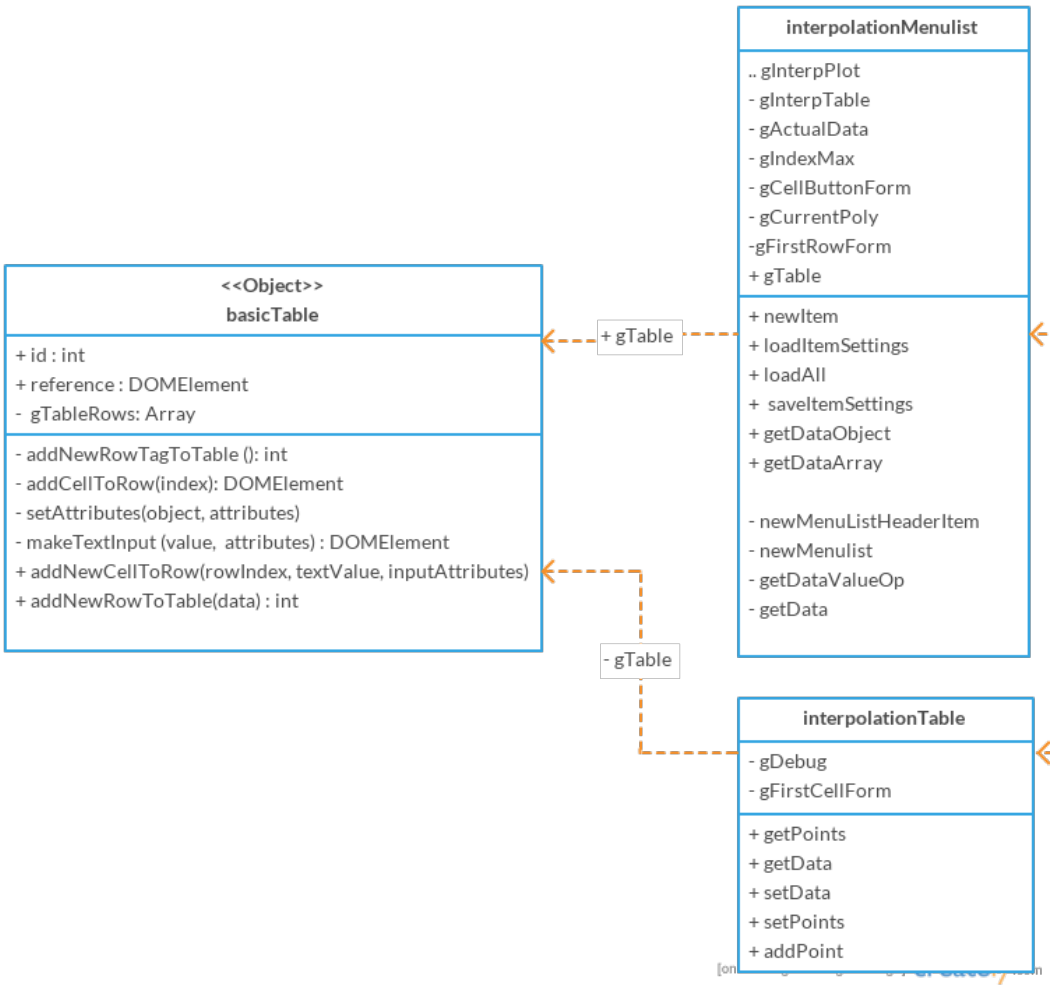
Új tábla létrehozása.

that.setCellForm((i , j, attributes))

Egy adott cella megformázás beállítása.

that.getNumOfCols()

Visszatér az oszlopok számával.



3.7. ábra. Táblázatok osztálydiagramja

that.getNumOfRows()

Visszatér a sorok számával.

that.getRow(i)

Visszatér a sor DOM-elemével az index alapján.

Ha nincs olyan indexű akkor null-al tér vissza.

that.getInputTag(i, j)

Visszatér a tábla input elemével.

that.getValue(i, j)

Egy adott cella érték lekérdezése.

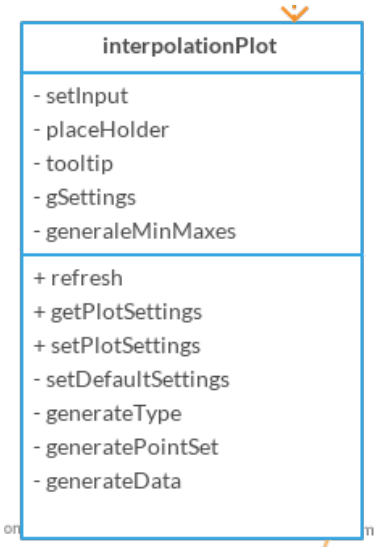
that.findValue(column, value)

Megkeresi melyik sorban van egy adott értéket.

that.setValue(i, j, value, form)

Beállít egy adott értéket egy cellának.

that.deleteTable()



3.8. ábra. Grafikon kirajzoló osztálydiagramja

Teljesen törli a táblázatot.

that.remove(row)

Kivesz egy sort a táblázatból.

addNewRowTagToTable ()

Ad egy új sort a táblázathoz.

addCellToRow(index)

Ad egy cellát a sorhoz.

setAttributes(object, attributes)

Beállítja egy objektum tulajdonságait.

makeTextInput (value, attributes)

TextInput hozzáadása a sorhoz.

interpolationMenulist (aConfig)

Az interpolációs menü függvénye. Itt tarjuk számon az aktuálisan betöltött adathalmazt.

that.newItem()

Új elemet vesz fel a listába. Gomb hatására is meghívódhat.

Új lista elem egy új interpolációs adathalmaz felvételét jelenti.

that.getDataArray(server)

Visszatér az adathalmazzal, tömb formában. Ebben a formában küldjük fel a szervernek.

that.getDataObject()

Eredményül adja az adathalmazt, egy objektum formájában. Az Objektum értékeinek kulcsa, az interpolációk egyedi azonosítója (id-ja).

that.saveItemSettings()

Elmenti az adatokat az aktuálisan kijelölt sorba.

that.loadItemSettings(index)

Feldogozza az adatsort a táblából, és betölti az adatokat a táblába.

that.loadAll(savedObject, resultObject)

Betölti az összes Interpolációt az adott adathalmazból

newMenulist()

Új menülista: régi menü kitörlése, és egy új generálása

interpolationPlot (aConfig)

Grafikon megjelenítése: "Flot" segítségével létrehoztam az alábbi Objektumot. Ebben valósítottam meg a kirajzolást, és annak tulajdonságait.

that.refresh(points, polynomials)

Pontok és a polinomok alapján frissíti a grafikont.

that.getPlotSettings

Visszatér a grafikon megjelenítési tulajdonságokkal. Ennek segítségével mentünk.

that.setPlotSettings

Betölti a grafikon megjelenítési tulajdonságokat.

generateData(senderData, polynomial)

Legenerálja a grafikon azon bemenő paraméterét, amely a megjelenítendő adatokat állítja.

generateType()

Legenerálja a grafikon azon bemenő paraméterét, amely a grafikon megjelenítését állítja.

setDefaultSettings()

Legenerálja a grafikon azon bemenő paraméterét, amely a grafikon megjelenítését állítja.

generatePointSet(tableArray, derivNum)

Legenerálja az adott pontokat, az interpolációs táblázatból.

interpolationTable (aConfig)

Az interpolációs Táblázat logikája, és generálása. Ebben a táblázatban tekinthetjük meg a pontokat.

that.addPoint(x, y, dn)

Hozzá adja a pontot a táblázathoz. Ha létezik ezen az X-en pont akkor frissíti.

that.setPoints(tableArray)
Feltölti a táblázatot egy adott tömb értékeivel.

that.setData(data)
Feltölti az adatokkal a táblát.

that.getData()
Visszatér a táblázatban szereplő adatokkal.

that.getPoints()
Visszatér a táblázatban szereplő pontokkal.

3.5. Elosztott rendszer megvalósítása

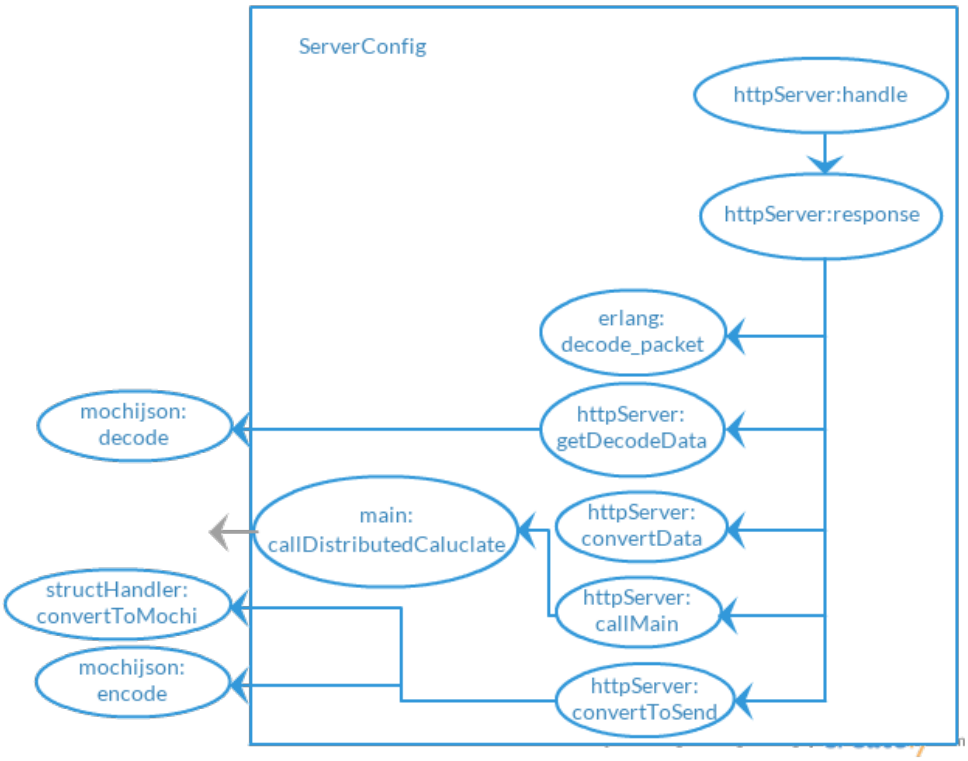
Elosztott rendszer Erlang-ban lett megvalósítva. Az elosztást interpolációnként végezzük, vagyis annyi processzt hozunk létre amennyi interpolációt kívánunk egyszerre kiszámítani.
A szerver figyel egy portot hogy érkezett-e rá adat. Ha érkezett adat az adott portra, azt kibontja, és elvégzi a szükséges műveleteket. Kinyer belőle egy listát mely az interpolálni kívánt pontokat és tulajdonságokat tartalmazza.
Tudjuk pontosan hány eleme van a listának, és annyi processzt hozunk létre. Ha vannak felcsatlakozva node-ok akkor a processzt az adott node-on is meg tudja hívni. Ha létrehozta a processzeket lista elemein végig megy, és azokat szétküldi a processzeknek, majd megvárja míg az összes végig ér, és visszatérve megkapja az eredményt.

3.5.1. Web-szerver kommunikáció

A webszerver kommunikációhoz a fájlok a httpServer.erl fájlban találhatóak meg. Az ebben található függvényeket a main-ben hívjuk meg amikor inicializáljuk a szervert.

httpServer:start(Port, WatcherNode)
Elindítja a szervert, az adott porton.
Ezt a függvényt a main-ben hívjuk, meg ahol már megkapja a node-figyelő pid-jét és az alapértelmezett port-ot

httpServer:response(Str, WatcherNode)
Miután érkezik egy kérés a szervernek ebben a függvényben kezeljük le. Innen indul ki minden folyamat ami a számítást végzi.
Az alábbi sorrendben hívódnak meg a függvények:
getDecodeData, convertData, callMain, convertToSend



3.9. ábra. Szerver hívási folyamata

httpServer:getDecodeData(_)

Visszatér a szervernek küldött paraméterrel

httpServer:convertData(ResponseParams)

Létrehozza a kapott adatból az Erlang struktúrát

httpServer:callMain(RespJson, WatcherNode)

Amikor az adatokat feldolgoztuk és minden rendben ment, elindítjuk a main függvényét, ezen függvény segítségével.

httpServer:convertToSend(Object)

Amikor a számítás véget ért, létrehozzunk a visszaküldéshez szükséges adatstruktúrát, majd elküldjük a szervernek.

3.5.2. Adat feldolgozás

Az adatot JSON-ben kapja a szerver. Az adathalmaz kibontásához MonchiJSON lett alkalmazva. A segédfüggvények és konvertálók a "Utility/structHandler.erl" fájlban lettek megvalósítva.

Elsősorban a megkapott speciális adathalmaz kibontására használtak az itt lévő függvények, de egyéb segédfüggvények is megtalálhatóak ebben a fájlban, amelyek a konvertálással kapcsolatosak.

Mochi-json kibontásához használt segédfüggvények:

structHandler:getElementByKeyList(KeyList, DataSetElement)

Visszatér egy értékkel, amely az adott kulcson van, ha egy elemű a kulcs lista. Több elem esetén a kulcsokban lévő értékeket nézi, és visszaadja a legbelső kulcson lévő elemet.

structHandler:getElementByKey()

Visszatér egy objektumban az adott kulcson lévő értékkel

Adat Struktúra az interpoláció meghívásához

structHandler:getDataByJson(JsonSting)

A mochi-json dekódoló meghívása, visszatér egy Erlang struktúrával.

structHandler:getDataSet(Data)

Visszatér az adatok halmazával. Ebből a halmazon, vagyis listán kell végig menni, és szétosztani az elemeit.

structHandler:getPoints

Pontok visszanyerése egy speciális módon, melyet a "calulator" fel tud használni

```
EmptyStruct =  [{x, []}, {y, []}]
```

structHandler:<Számítási paraméterek>

getInverse(DataSetElement) - inverz-e
getType(DataSetElement) mi a típusa?
getId(DataSetElement) egyedi azonosítója
getPoints(DataSetElement) pontok struktúrája

Az eredmény visszanyeréséhez az alábbi segédfüggvényeket kellett használni:

structHandler:convertToMochi(Object)

A mochi-json Erlang struktúra annyira nem egyértelmű elemekből áll. Speciálisan kell felépíteni az eredményt. Ez a függvény megkap egy Erlang listát és átkonvertálja mochi-json-nak megfelelő struktúrává, majd átkonvertálja egy json string-gé.

structHandler:simplifyPolinomial(Result, Array)

Egyszerűsíti a polinomot amelyet eredményül kapott.

3.5.3. Gép-szerver kommunikáció

pidWatch:startPidWatch()

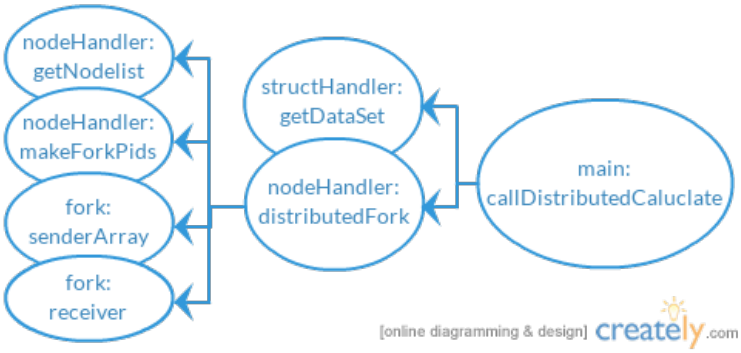
Elindítja a node-figyelőt, melyben feliratkozni lehet a listára, vagy lekérdezni az adatokat. A node-figyelő indulás után figyelni fog és ha küldenek neki egy kérést, akkor azt kezeli.

pidWatch:registerToServer(Pong_Node)

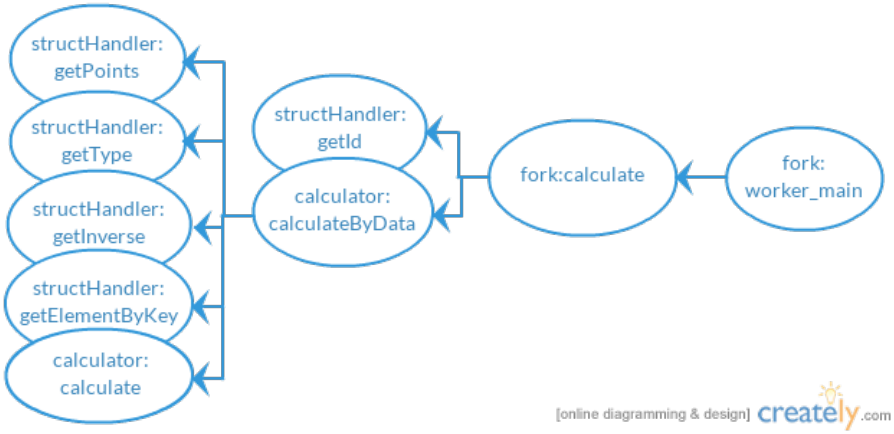
Ezzel a kéréssel lehet felcsatlakozni a szerverre. A kérést elküldi és ha sikeres volt a feliratkozás, akkor ok-al tér vissza.

3.5.4. Elosztás megvalósítása

Az elosztást tartalmazó fájlokat a /DistributedSystem mappában találjuk meg. nodeHandler.erl fájlban találhatóak a processz létrehozással kapcsolatos függvények. fork.erl fájlban a processz kommunikáció logikája van megvalósítva.



3.10. ábra. Elosztás folyamat hívásai



3.11. ábra. Gyerek folyamat hívásai

nodeHandler:distributedFork(NumOfPids, DataList, WatcherNode)
Létrehozza a számításhoz szükséges Node Struktúrát. LogicModule-ban szereplő senderstart, recivestart, worker_main

nodeHandler:getNodelist
Lekéri a node-figyelőtől a felcsatlakozott node-okat.

nodeHandler:makeForkPids
Létrehozza a számításhoz szükséges új processzeket.

fork:senderArray
Végig megy egy adott tömbön és az elemeit szétküldi a processzeknek.

fork:receiver
Válaszok érkezésére vár a processzektől. Ha minden válasz megérkezett, akkor visszatér.

fork:worker_main
A gyerek processzek függvénye. Várja a szülőtől az adatot, számol vele és visszaküldi.

fork:calculate
A fork-ból a számítást hívó függvény. Eredménnyel visszatér, majd azt olyan formára hozza, amit vár a szülő.

3.6. Kalkulátor

A Kalkulátor részben számítódik ki egy-egy interpolációnak az eredménye. A megkapott adatok alapján számol, ha kell létre hozza a kezdő mátrixot, kiszámolja az eredmény mátrixot, majd annak segítségével kiszámolja a polinomot.

3.6.1. Felépítés

A számítást végző rész, nem áll sok fájlból, ezért ez nem lett sok részre szétbontva.

calculator.cpp
Az egész számítás itt van megvalósítva, minden függvény, és segédfüggvény is.

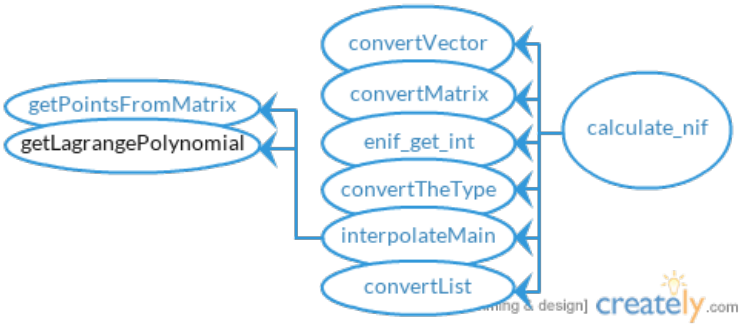
erlang.cpp
Az Erlang-gal való kommunikáció megvalósítása

main.cpp
C++-os modul különálló tesztelésére kellett.

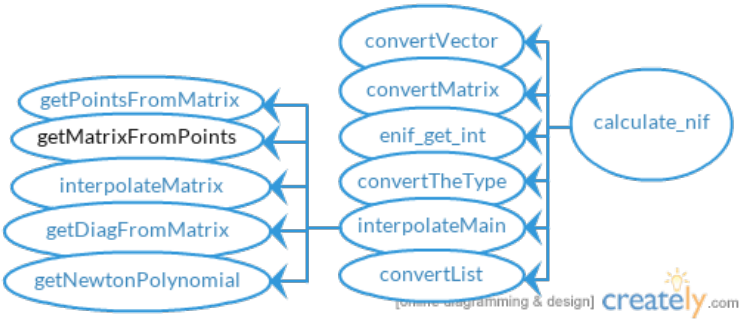
logTest.cpp

Teszt függvények, melyekben dinamikus tesztesetek és paraméterezhető tesztesetek is vannak.

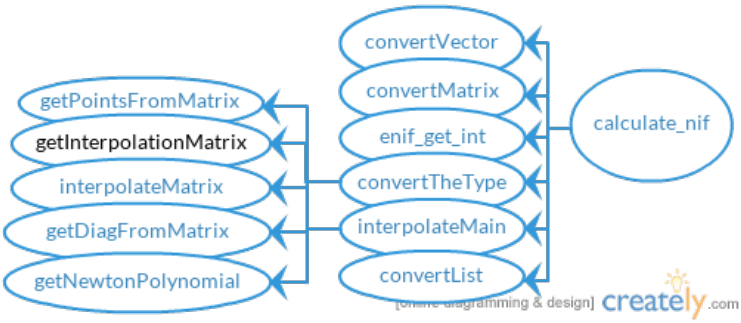
3.6.2. Fontosabb számítási függvények



3.12. ábra. Lagrange számítás hívási folyamata



3.13. ábra. Newton számítás hívási folyamata



3.14. ábra. Hermite számítás hívási folyamata

DArray interpolateMain

Kívülről meghívandó fő függvény mely elosztja és konvertálja a részeket

DArray &x : Az x pontok listája

DMatrix &Y : Az x pontokhoz tartozó y pontok halmaza

string type : Interpoláció típusa: Lagrange, Newton, Hermite

bool inverse : Inverz interpoláció kell-e

void interpolateMatrix(DArray &x, DMatrix &M)
Interpolációs Táblázat kiszámítása

DArray l(int j, DArray X))
Lagrange polinom számítás segédfüggvénye

DArray getLagrangePolinomyal(DArray X, DArray Y)
Lagrange polinom számítás

DArray omega(int j, DArray X)
Newton polinom számítás segédfüggvénye

DArray polynomialAddition(DArray P, DArray Q)
polinom összeadás

DArray polynomialMultiply(DArray P, DArray Q)
polinom szorzás

DArray getPointsFromMatrix(DMatrix Y)
Pontokat(0. derivált) visszaadja a mátrixból

DMatrix getMatrixFromPoints(DArray Y)
Mátrixot ad vissza a pontokból

DArray getDiagFromMatrix (DMatrix &M)
Diagonális lekérése a mátrixból

void getInterpolationMatrix
(DArray X, DMatrix Y, DArray &resX, DMatrix &resM)
X és Y ponthalmazból visszaadja a mátrixot

3.6.3. Elosztott rendszerrel való kommunikáció

Az elosztott rendszerben hívódó számítást Erlang - erl_nif"-el sikerült megoldanom. Az ezzel kapcsolatos dolgokat az Calculator/erlang.cpp tartalmazza.

static ERL_NIF_TERM calculate_nif
Ennek a függvénynek a segítségével valósul meg a kettő közötti kommunikáció

static int convertVector
Erlang lista C++ vektorrá konvertálása

static int convertMatrix

Erlang lista lista konvertálása C++ vektor vektorrá

static ERL_NIF_TERM convertList

Erlang listává konvertálás egy C++ vektorból

convertTheType

típus számot konvertálja string-gé: "newton", "hermite", "lagrange"

static ErlNifFunc nif_funcs

Felsorolja milyen függvényeket importálunk az Erlang-ba

Calculator/calculator.erl fájl tartalmazza az alábbi függvényeket:

calculator:init()

"erlang:load_nif" segítségével betölti a lefordított c++ fájlt.

calculator:calculate(_X, _Y, _Type, _Inverz)

Erre a függvényre lesz ráfordítva a C++-os függvény.

calculator:calculateByData(DataSetElement)

A bejövő paraméterből kinyeri a pontokat és a típust, majd meghívja a számító függvényt.

3.7. Tesztelési terv

A felület manuális teszteléssel lett kipróbálva, automatizált tesztesetek nem lesznek.

3.7.1. Kalkulátor tesztelés

Ebben a részben a cpp-ben írt tesztesetekről lesz szó.
A tesztelést folyamatosan végeztem a minta adatok alapján. A függvények implementálása közben ezekre a minta adatokra meghívtam, majd ezekkel számoltam. A teszteléshez a logTest.cpp fájlban található függvényeket alkalmaztam.
A fájlban a javítást segítő kiíratók függvények, integrációs teszt esetek, valamint manuális, felület nélküli számítást végző tesztesetek vannak.
Ezeket a teszteket nagyrészt kiváltották a szerveren futó tesztek, így fejlesztésük abba maradt, előfordulhat hogy a tesztek elavultak.

bool testAll()

Minden teszt lefuttatása. Ha minden teszt jól futott le akkor "true" értékkel tér vissza.

void testInterpolation(bool logPoly = false)
Interpolációs teszteket futtat. Mindegyik teszt egy egyszerű interpolációt vesz alapul, és a számítás eredményét ellenőrzi.

bool testMainInterpolation(bool logPoly = false)
Fő függvény tesztje. Elsősorban a feltétel átmeneteket teszteli, és azt hogy valóban számol-e ezen keresztül interpolációt.

bool testNewton(bool logPoly = false)
Newton számítás tesztje.

bool testLagrange(bool logPoly = false)
Lagrange interpoláció tesztje.

void testPolynomial()
Interpolációs mátrix tesztje.

void testMatrixInterpolation()
Manuális mátrix számítás függvénye.

void testManualInterpolation()
Manuális interpoláció számítás függvénye.

testManualPolynomial()
Manuális polinom összeadás és szorzás függvénye.

void genXSquaredPoints(DArray &X, DMatrix &Y)
generál egy minta X,Y ponthalmazt az x^2 pontjaiból. testMatrixInterpolation Segédfüggvénye feltölti az x^2 pontjaival.

3.7.2. Komponens és integrációs tesztelés

Főként az elosztást és a párhuzamosítást, valamint a szerveren lévő adatfeldolgozást teszteltem ilyen formában. Ezekkel a tesztesetekkel ellenőriztem a szerver és a számítás helyességét, és a szerver futást, miközben fejlesztettem.
Ezeket a teszteket érdemes lefuttatni, amikor a szervert konfiguráljuk. A szerveren lévő komponensek és azok egymással való kommunikációja fut le ilyenkor. Ha a teszten átmennek, akkor nagy valószínűséggel a szerveren már nem lehet probléma.
Viszont ha egy modul rosszul van leforgatva, vagy nincs betöltve, a tesztek hibát jeleznek. Ha a szerveren vagy gépen nem sikerült a tesztek lefutása, nem lehetséges a számítás elvégzése. Ennek oka valamelyik modul rosszul való betöltése, vagy az Erlang, GCC verziója nem megfelelő a számításokhoz.
Az átfogó tesztelés a ServerConfig/test.erl fájlban található.

test:fork

Teszt futtatása a fork-nak ez a teszteset az párhuzamosítás miatt volt fontos. Viszont egy másik teszt átvette a helyét.
Ha mégis szükség lenne az általános párhuzamosítás tesztelésére, ezt kellene továbbfejleszteni.

test:runCheck, run

Futtatást kezelő függvények Lefuttatják azokat a teszteket amiket a ServerConfig/main.erl-ből már el lehet indítani. A main-ben található függvény tesztelésére a test:simulateDistribute függvényt használjuk, amit a bin/run.erl-ben lévő tesztesetek

test:simulateDistributedCalculate

Egy olyan minta adatból, melyet a szerver is küldhet, elvégzi a számítást és ellenőrzi az eredményt.
Ha megadjuk neki a node-figyelő pid-jét akkor elosztott számítást is teszteli, és a node-figyelővel való kommunikációt.

test:simplifyPolinomialTest

Ellenőrzi a struktúra kezelőben megvalósított polinom egyszerűsítést.

test:convertMochiElements

structHandler: getTableData és getElementByKey tesztelésére írt függvény.

test:getResultTest

Eredmény konvertálásának tesztje. Szimulál egy processzektől visszakapható eredményt, majd ezt átalakítja a küldéshez megfelelő formátummá.

test:getParseJSONParams

Json string-ből a számításhoz szükséges minden paraméter kinyerésének tesztelése (structHandler teszt).

test:convertStruct

structHandler függvényeinek tesztje: getNewPointStruct, appendNewPointStruct, convertPoints

test:simulateFirstParseAndRun

Kibontja az első elemet a mintából, és számítás után ellenőrzi az eredmény helyességét.
Ez a teszt a számítást és az adat konvertálást teszteli, a párhuzamosítást/elosztást nem.

test:getFirstElementOfDataSet

Visszaadja a minta adatok első elemét.

test:getJSONString

Egy minta adathalmazt ad vissza (json string) ami jöhet a felületről.

test:getResultTestHelper

Számítási eredményekből és az elvárt eredményből a tesztelés eredményt állítja elő.

3.7.3. Manuális tesztelés

Elsősorban a felület átfogó tesztelését végeztem ilyen módon, de a szerverrel való kommunikációnál és az elosztásnál is előkerültek ilyen módon hibák.
Az alábbi táblázatban felsoroltam a hibákat, melyek feltűntek a tesztelés során.

Hiba	Javítva	Info
Szerver hiba bizonyos mennyiségű adatküldése felett	nem	Előidézés: 9-nél több egyszerű adat felküldése, minta nagy adatból 4db felett. A szerver egybe küldi az értékeket, a httpServer modul már nem kapja meg a végét. Nem küldi 2 részletben, ezt kizártuk.
Hermite inverz nincs implementálva, és mégis beállítható a felületen.	nem	El tudjuk küldeni az oldalról úgy az adatokat hogy Hermite interpolációnál is állítható az inverz, közben a szerver nem inverz interpolációt fog számolni.
Ha kilép egy node, akkor elszáll a számítás	nem	Ötletek: node-ok kivételére is kellene opció,vagy kezelni kellene az elszálló node-okat.
Node Lista lekérdezésnél valamikor végtelen ciklusba fut a lekérdezés.	nem	Hiba: túl sokszor küldjük el ugyanazt az üzenetet, és kapunk vissza rossz választ, kéne bele egy időkorlát is, a várakozásra.
Nem jó pontosság esetén nem jelenik meg a polinom	igen	Előidézés: beírsz betűket a pontossághoz
Egy interpoláció adatbetöltési hibák	nem	Egy interpoláció tulajdonságainak betöltésénél nem állítja be az interpoláció típust és azt hogy inverz-e. Nem jelenítjük meg a polinomokat sem bármilyen egyéb formában.
Eredmény betöltésnél előfordul hogy a pontosság undefined	nem	Előidézés: ? Minta adatban fordul csak elő, de ha onnan is betöltődhet akkor máshonnan is, amikor betölti az adatokat le kellene ezt kezelni.

Irodalomjegyzék

[1] Gergó Lajos: Numerikus Módszerek, ELTE EÖTVÖS KIADÓ, 2010, [329], ISBN 978 963 312 034 7

[2] http://www.erlang.org/doc/man/erl_nif.html 2015

[3] http://stackoverflow.com/questions/2206933/how-to-write-a-simple-webserver-in-erlang 2015

[4] http://erlang.org/doc/man/gen_tcp.html 2015

[5] https://www.sharelatex.com/learn/Sections_and_chapters 2015

[6] http://tex.stackexchange.com/questions/137055/lstlisting-syntax-highlighting-for-c-like-in-editor 2015