FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Simulační a evaluační prostředí pro řízení kontinuální chromatografie
Student:	Adam Svoboda
Vedoucí:	Ing. Svatopluk Henke, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2021/22

Pokyny pro vypracování

1. Vypracujte literární rešerši na téma řešičů matematických modelů chromatografie.
2. Prozkoumejte dostupný volně šiřitelný i komerční software v této oblasti a zároveň možnosti jejich propojení na webová rozhraní.
3. V experimentální části proveďte v libovolném programovacím jazyku (C++, Python, Fortran, aj.) implementaci vybraného algoritmu pro řešení matematických modelů chromatografického procesu a případně jeho optimalizaci (modelové řízení či soft computing metody).
4. Navrhněte, implementujte a otestujte vhodné operátorské a simulační webové prostředí pro vizualizaci simulace chromatografického procesu, sledování chodu a ovládání vlastního separačního zařízení.
5. Proveďte experimenty a na základě výsledků diskutujte vhodnost použitých metod a porovnejte výsledky s údaji uvedenými v aktuálních literárních zdrojích.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 1. září 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Simulační a evaluační prostředí pro řízení kontinuální chromatografie

Adam Svoboda

Katedra . . .Katedra softwarového inženýrství
Vedoucí práce: Ing. Svatopluk Henke, Ph.D.

7. ledna 2021

Poděkování

Chtěl bych poděkovat svému vedoucímu panu Ing. Svatopluku Henke, Ph.D. za jeho vstřícnost a odbornou pomoc při zpracování této práce. Také bych chtěl poděkovat panu Ing. Tomáši Svobodovi za jeho cenné rady a ochotu, které byly velkou pomocí.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. ledna 2021 . . . . . .

Odkaz na tuto práci

Svoboda, Adam. *Simulační a evaluační prostředí pro řízení kontinuální chromatografie*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá vytvořením webové aplikace pro simulaci chromatografických procesů. Nejprve se stručně zabývá chromatografií, její historií, základními principy a druhy. Dále se zabývá základními chromatografickými modely, jejich řešením a použití při vytváření simulací. Nakonec se zabývá samotnou aplikací, použitými nástroji, algoritmem simulace chromatografie, komunikace klienta se serverem a tvořením stránek pro zobrazení simulace.

Klíčová slova Chromatografie, Simulace, Webová aplikace

Abstract

This paper deals with creating web application for simulation of chromatographic processes. First, it briefly discuss chromatography, its history, fundamental principles and types. Next, it deals with basic chromatography models, solving of these models and their application in creating simulations. Last, it talks about creating the application, used tools, algorithm of chromatography simulation, communication between client and server and creation of pages for displaying the data.

Keywords Chromatography, Simulation, Web application

Obsah

Seznam obrázků

Úvod

Chromatografie je skupina separačních metod, založena na rozdílné rychlosti pohybu látek v soustavě stacionární a mobilní fáze. Když se mobilní fáze s rozpuštěnými složkami pohybuje stacionární fází, každá složka se pohybuje jinou rychlostí a dochází tak k jejich separaci.

Tento proces se používá k analýze látek a získávání cenných látek ze směsí. Má uplatnění v mnoha odvětvích průmyslu, od chemického až po potravinářský. Aby se však chromatografie dala použít v průmyslu, bylo třeba vymyslet, jak ji využít optimálně, ať už je problém cena, čistota produktu nebo jiné faktory. Proto bylo nutné vymyslet způsob modelování chromatografických procesů a jejich simulace. [3]

Jedna z nejpoužívanějších je chromatografie v režimu SMB (simulated moving bed), která umožňuje simulovat pohyb stacionární fáze periodickým přepínáním vstupních a výstupních proudů. Objevilo se proto mnoho společností, které se specializují na tento proces a nabízí poradenství při nastavování stanic tohoto typu. [4]

V této práci stručně popíšu některé důležité druhy chromatografie, jejich princip a použití. Dále se budu zabývat modelováním chromatografických procesů, jaké modely existují a na čem jsou založeny. Nakonec se budu zabývat jejich simulací a optimalizací. Také se budu zabývat již existujícím softwarem a jeho případnými nedostatky.

V praktické části vytvořím webovou aplikaci, která bude simulovat chromatografický proces na základě některých modelů, konkrétně jeden jednoduchý diskontinuální proces a jeden proces v režimu SMB, a dále bude umět ovládat vzdálenou chomatografickou stanici.

Kapitola 1

Cíl práce

Cílem této práce je zaprvé stručně informovat čtenáře o tématu chromatografie, popsat její historii a vývoj, vysvětlit její důležitost, například při výrobě ibuprofenu, separaci vitamínů a separaci ropných frakcí. [4] Dále nahlédnu do tématu simulace chromatografie, což je důležitá část zavádění chromatografických procesů do průmyslové výroby a zároveň je to hlavní téma praktické části této práce.

Dalším cílem bude samotný projekt, webová aplikace, která bude umožňovat simulaci chromatografického procesu. Budu popisovat jaké nástroje jsem použil při vývoji, problémy na které jsem narazil a výsledný produkt.

Nakonec budu diskutovat o postupech, které jsem použil a o jejich vhodnosti, a závěrech, ke kterým jsem došel při tvorbě této práce.

Kapitola 2

Analýza a návrh

2.1 Chromatografie

Jako chromatografie se označuje skupina separačních chemicko-fyzikálních procesů, při které se směsi látek v rozpouštědle pohybuje v určitém prostředí. Kvůli rozdílným vlastnostem jednotlivých látek se každá pohybuje jinou rychlostí a dochází tak k jejich separaci. Roztok látek, které separujeme, se nazývá mobilní fáze, a prostředí, ve kterém se roztok pohybuje se nazývá stacionární fáze. [5]

Existuje mnoho druhů chromatografie, které se dělí do několika kategorií. Chromatografie se mohou dělit podle vlastnosti látek, kvůli kterým k separaci dochází, například adsorpční chromatografie, kdy se různé látky zachycují v adsorbentu více než jiné a dochází tak k jejich zdržení. Dále se chromatografie dělí podle skupenství mobilní fáze na kapalinovou a plynovou, podle uspořádání stacionární fáze, například kolonová chromatografie, podle účelu na analytickou či preparativní a podle průběhu na diskontinuální a kontinuální. [5]

2.1.1 Historie chromatografie

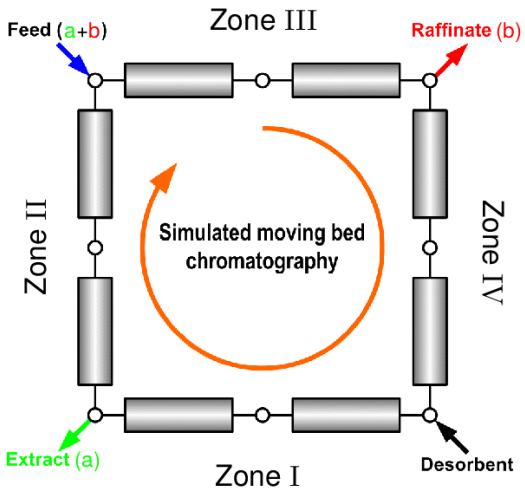
Objev chromatografie je připisován Mikhailu Tsvetovi, ruskému botanikovi, který v roce 1900 prováděl výzkum o rostlinných pigmentech a chromatografii použil k jejich separaci. [6]

Rozvoj chromatografie byl však velmi pomalý, protože tehdejší procesy byly časově velmi náročné a citlivé na okolní podmínky a neexistovaly technologie pro zlepšení efektivity. Velkého rozmachu se chromatografie dočkala až v polovině 20. století, kdy technologický rozvoj umožnil vývoj procesů a přístrojů. V dnešní době se chromatografie běžně používá v chemickém a farmaceutickém průmyslu k analýze a výrobě, nově se začíná chromatografie používat v potravinářském průmyslu k výrobě cukrů a aminokyselin. [4]

2.1.2 Kontinuální chromatografie

Dosud běžné chromatografické procesy byly diskontinuální, není možný nepřetržitý průtok mobilní fáze, což je velmi nepraktické, když je třeba separovat větší množství látky.

Jedním z významných objevů byl proto vynález kontinuální chromatografie, která umožňuje separační proces provádět nepřetržitě. Toho lze dosáhnout pohybem stacionární fáze v opačném směru fáze mobilní, tzv. true moving bed (TMB). Pomalejší složka látky je unášena stacionární fází opačným směrem, zatímco rychlejší složka se pohybuje směrem mobilní fáze, což umožňuje nepřetržitou separaci. Z praktického hlediska je však TMB chromatografie téměř nerealizovatelná, je velmi složité zajistit plynulí protichůdný pohyb stacionární fáze. Byla proto vyvinuta tzv. simulated moving bed (SMB) chromatografie, ve které nedochází ke skutečnému pohybu stacionární fáze. Tento pohyb je simulován přepínáním čerpadel v směru pohybu mobilní fáze. [4]



Obrázek 2.1: Schéma SMB chromatografie [1]

2.1.3 Modelování chromatografie

Jako u mnoha jiných fyzikálních procesů, abychom je mohly zkoumat a efektivně využít, je vhodné vytvořit matematický model, který tento proces popisuje. Pro chromatografii existuje několik takových modelů s různým stupněm přesnosti a složitosti.

U modelovaní chromatografie založené na adsorbci je potřeba si zvolit adsorbční izotermu, křivku, která popisuje závislost koncentrace látky v adsor-

bentu na koncentraci látky v rozpouštědle za konstantní teploty. [7]

Těchto izoterem existuje několik, například lineární izoterma, která uvažuje ideální adsorbci, má proto lineární závislost a není vhodná pro větší koncentrace.

$$q^*_{i,s} = K_{H,i} c_{i,l}$$
$$0 < K_{H,i} < 1$$

Dalším příkladem je Langmuirova izoterma, která bere v úvahu kapacitu sorbentu. Proto při vyšších koncentracích konverguje k limitě a lze ji použít i pro vyšší koncentrace.

$$q^*_{i,s} = \frac{Q_{i,max} K_{L,i} c_{i,l}}{1 + K_{L,i} c_{i,l}}$$

Samotné modely pak nejčastěji popisují chromatografii parciálními diferenciálními rovnicemi. Základním modelem je Equilibrium ideal model (EIM), který definoval Don De Vault v roce 1943. Tento model přes jeho jednoduchost často dává dobré výsledky při použití nelineárních izoterem a je proto běžně používaný. [8]

$$\frac{\partial c_{l,i}}{\partial t} = -u_m \frac{\partial c_{l,i}}{\partial x} - \frac{1-\varepsilon}{\varepsilon} \frac{\partial q^*_{s,i}}{\partial t}$$

Počáteční podmínky:

$$c_{l,i}|_{t=0} = 0$$

Okrajová podmínka:

$$\left(\frac{\partial c_{l,i}}{\partial x} \right) \Big|_{x=0} = u_m (c_{l,i} - c^{in})$$

Složitější alternativou je potom Equilibrium dispersive model (EDM), který přidává dispersní člen. Tento model je vhodný pro systémy, kde interakce složek se sorbentem nejsou příliš silné. [9]

$$\frac{\partial c_{l,i}}{\partial t} = -u_m \frac{\partial c_{l,i}}{\partial x} + D_{ax,i} \frac{\partial^2 c_{l,i}}{\partial x^2} - \frac{1-\varepsilon}{\varepsilon} \frac{\partial q^*_{s,i}}{\partial t}$$

Počáteční podmínky:

$$c_{l,i}|_{t=0} = 0$$

Okrajové podmínky:

$$\left(\frac{\partial c_{l,i}}{\partial x} \right) \Big|_{x=0} = \frac{u_m}{D_{ax,i}} (c_{l,i} - c^{in})$$
$$\left(\frac{\partial c_{l,i}}{\partial x} \right) \Big|_{x=L} = 0$$

2.1.4 Simulace chromatografie

Protože existují matematické modely pro chromatografii, můžeme vytvářet simulace a předvídat chování systému při zadaných podmínkách. Simulovat chromatografický systém pro nás tedy znamená vyřešit parciální diferenciální rovnici (PDR) modelu, který jsme se rozhodli použít.

Prvním způsobem řešení parciálních diferenciálních rovnic je řešení analytické. Najít analytické řešení není ovšem vždy možné, a i když možné je, tak je velmi složité a často vyžaduje nějaký předpoklad, a je proto nepraktické z hlediska řešení těchto rovnic pomocí počítače. V dnešní počítačové době je proto mnohem vhodnější si parciální diferenciální rovnici diskretizovat. To znamená, že si náš systém časově a prostorově rozdělím na konečné množství bodů, tzv. časový krok a prostorový krok. [10]

Pro vytvoření diskretizovaného systému použijeme metodu konečných diferencí, při které derivaci v daném bodě nahrahdíme vhodnou diferenční formulí. Pro tuto aproximaci je možné použít dopřednou diferenci, kdy derivaci v daném bodě aproximuji pomocí následujících bodů, zpětnou diferenci, kdy aproximuji pomocí bodů předešlých, nebo centrální diferenci, kdy aproximuji pomocí bodů z obou stran. Volba často závisí na počátečních a okrajových podmínkách. [11]

Pro naše řešení použijeme metodu konečných diferencí. Z okrajových podmínek víme, jak se systém chová na začátku systému a díky počátečním podmínkám definujeme, jak systém vypadá na začátku. Jednotlivé časové kroky jsme pak schopni aproximovat z kroků předešlých použitím zpětné diference, konkrétně použijeme Wendroffovu formuli[12]:

$$c_{i+1}^{j+1} = c_i^j + \frac{1-ap}{1+ap}(c_{i+1}^j - c_i^{j+1})$$

Kde:

$$p = k/h$$

$$a = u_m \frac{\varepsilon_t}{\varepsilon_t + (1-\varepsilon_t)K}$$

$$0 \leq ap \leq 1$$

Nástřik je standardně definován skokovou funkcí, kdy se koncentrace v čase začátku a konce nástřiku změní okamžitě. Při reálném experimentu není možné okamžitého skoku dosáhnout. Proto jsem definoval tzv. nástřikovou funkci, která na začátku a konci nástřiku popisuje chybovou funkci. Takto definovaný nástřik byl použit pouze u simulace diskontinuální chromatografie. Výhodou tohoto postupu je kromě eliminace numerické disperze také přiblížení se průběhu reálného experimentu.

Pro začátek nástřiku:

$$c(x,t) = \frac{c_{feed}}{2}\left[1 + \frac{t - t_{feedstart}}{4\sigma * \sqrt{2}}\right]$$

Pro konec nástřiku:

$$c(x,t) = \frac{c_{feed}}{2}\left[1 - \frac{t - t_{feedstart}}{4\sigma * \sqrt{2}}\right]$$

2.2 Specifikace projektu

Cílem je vytvořit webovou aplikaci, která bude umět simulovat průběh chromatografie na základě parametrů, které zadáme. Aplikace bude mít dvě hlavní části, jednu pro simulaci diskontinuální chromatografie a druhou pro simulaci kontinuální chromatorafie v režimu SMB. Tyto procesy vhodně graficky vykreslí.

2.2.1 Potřebné nástroje

Protože se jedná o webovou aplikaci, nejdříve je třeba webový framework, který ulehčí práci s webovými protokoly a umožní komunikaci mezi serverovou a klientskou částí aplikace. Těchto frameworků existuje mnoho a téměř pro každý moderní programovací jazyk máme na výběr z několika možností. Mezi používanější patří Laravel pro PHP, Spring pro Javu, ASP.NET Core pro C#, Django pro Python a Express.js pro Javascript.[13]

Dále budou použity tyto balíčky:

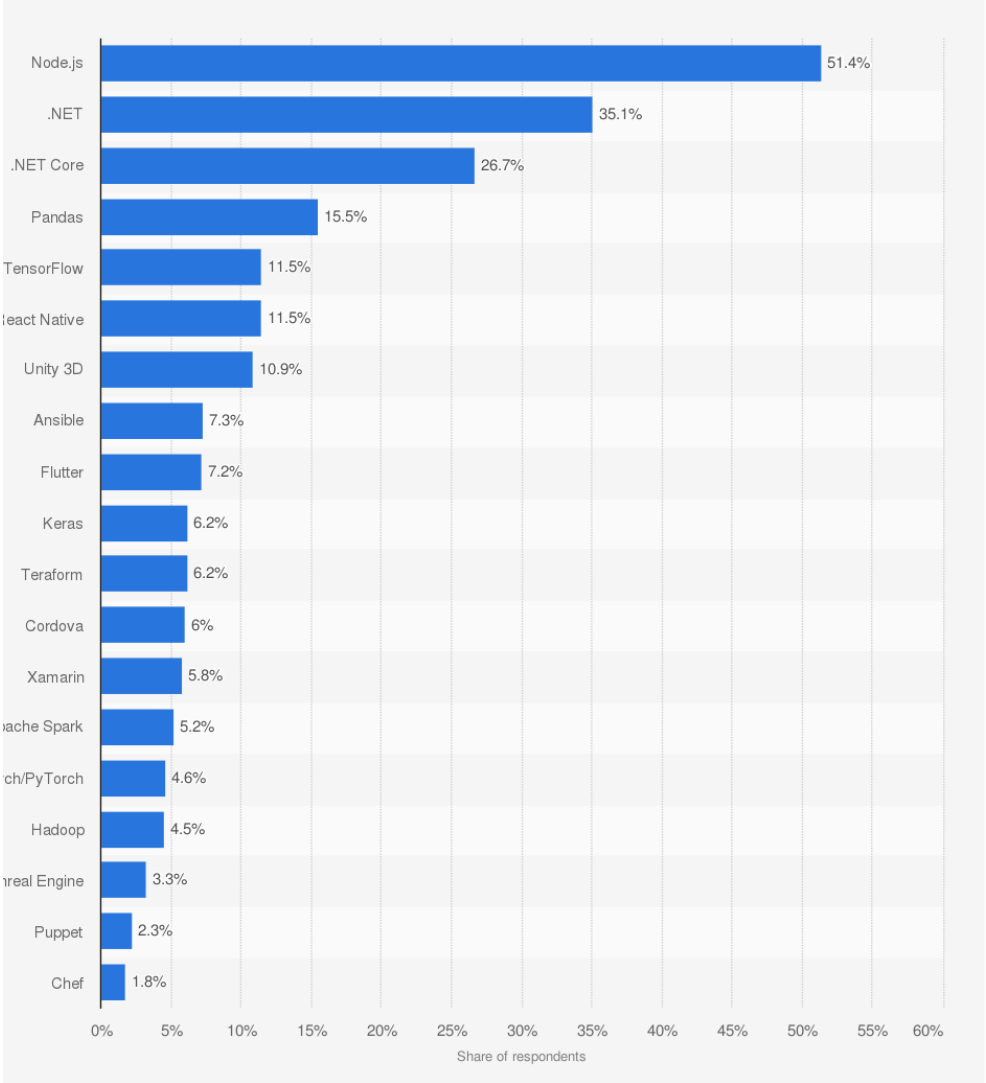
- Matematický balíček, pro některé matematické funkce.
- Balíček pro vykreslování grafů.
- Balíček pro generování unikátních ID.
- Balíček pro deserializaci těla requestů.

Nakonec je třeba nástroj pro verzování, například SVN, CSV nebo GIT. Tyto nástroje jsou často velmi rozvinuté a mají mnoho vlastností, které podporují paralelní vývoj pro větší týmy vývojářů. Protože tuto aplikaci vyvíjím sám, budu tyto nástroje používat pouze pro verzování.

2.3 Výběr nástrojů

Výběr nástroje souvisí s výběrem programovacího jazyka. Protože klientská část mojí aplikace má být spouštěna v prohlížeči, bude použit Javascript. Rozhodl jsem se pro Javascript i pro serverovou část za použití Node.js.

Node.js je runtime prostředí pro Javascript, které umožňuje zpracovat a spouštět Javascriptový kód mimo prohlížeč, díky čemuž můžeme psát serverové a klientské části webových aplikací ve stejném jazyce. V posledním desetiletí se Node stal velice populární, což vedlo k jeho velkému rozvoji. Díky tomu má Node dnes velkou knihovnu balíčků a nástroj pro správu balíčků NPM (Node package manager), který velmi ulehčuje jejich stahování a použití.[14]



Obrázek 2.2: Nejvíce používané knihovny, frameworky a nástroje mezi developery [2]

Díky výběru programovacího jazyka je výběr webového frameworku snadný. I když možností je stále několik, Express.js se stal, díky jeho jednoduchosti, nejpopulárnějším frameworkem pro vývoj serveru webových aplikací pro Node.[15]

I výběr knihoven je značně ulehčen. Math.js je standardní matematický balíček, uuid je standardní balíček pro generování unikátních ID a body-parser je standardní balíček pro deserializaci.

Pro knihovnu na vykreslování grafů máme mnoho možností, lišící se svou složitostí a způsobem použití. Rozhodl jsem se pro ChartJS, která je dostatečně přizpůsobitelná pro naše potřeby, a zároveň není příliš složitá k použití.

Pro verzování jsem se rozhodl pro GIT, protože je to jediný verzovací nástroj, se kterým mám už nějaké zkušenosti, a protože se také stal standardem pro vývoj.

Kapitola 3

Realizace

3.1 Instalace a příprava

3.1.1 Node a potřebné balíčky

Prvním krokem při tvorbě serverové části je instalace potřebných nástrojů. Začal jsem stažením a instalací Node.js z oficiální stránky nodejs.org. Node je již distribuovaný s již zmiňovaným nástrojem NPM, takže jsem si založil pracovní složku, inicializoval jsem si NPM pomocí *npm init* a pomocí příkazu *npm install* jsem si stáhl všechny potřebné balíčky.

- *npm install express* pro instalaci webového frameworku express.
- *npm install mathjs* pro instalaci matematické knihovny mathjs.
- *npm install uuid* pro instalaci balíčku na generování unikátních ID.
- *npm install body-parser* pro instalaci balíčku pro deserializaci.
- *npm install path* pro instalaci knihovny, která ulehčuje práci s cestami k souborům.

Pro starší verze NPM se tyto příkazy musely volat s přepínačem -S nebo --save, aby se balíčky uložily jako závislosti pro náš projekt, od verze 5.0.0 je však ”save”chování dané jako výchozí a nemusíme přepínač použít.

Výsledkem je složka s balíčky a dva soubory, *package.json*, hlavní konfigurační soubor, a *package-lock.json*, který obsahuje informace o instalovaných balíčkách. Pro nás je důležitý *package.json*, pomocí kterého definujeme projekt a použité balíčky. Takto vypadá tento soubor pro náš porjekt

```
{
 "name": "chroma",
 "version": "1.0.0",
```

```
 "description": "test",
 "main": "index.js",
 "scripts": {
  "start": "node index",
  "dev": "nodemon index"
 },
 "author": "Adam Svoboda",
 "dependencies": {
  "body-parser": "^1.19.0",
  "express": "^4.17.1",
  "mathjs": "^7.5.1",
  "mysql": "^2.18.1",
  "uuid": "^8.3.1"
 },
 "devDependencies": {
  "nodemon": "^2.0.6"
 }
}
```

3.1.2 Git

Po úvodních testech jsem si pracovní složku inicializoval pro git pomocí *git init* a vytvořil jsem si nový projekt na školním gitlabu gitlab.fit.cvut.cz pojmenovaný **chroma**. Dále jsme do kořenové složky přidal soubor *.gitignore*, který mi zajistí, že se bude ignorovat složka s balíčky, která se běžně neverzuje. Lokální repozitář jsem si spároval se vzdáleným a pomocí *git remote add origin* a provedl jsem *innitial commit*.

3.2 Serverová část

Hlavní program serverové části bude index.js, který předá uživateli složku public, ve které budou veškeré HTML stránky, CSS styly a Javascripté kódy, které mají být spouštěny na straně klienta, a reaguje na requesty poslané klientem.

Nejprve musíme importovat express pomocí příkazu require.

```
const express = require('express');
const app = express();
```

Nyní můžeme začít definovat requesty. Express používá tento syntax:

```
app.METHOD('/ROUTE', FUNTION)
```

Za METHOD doplníme HTTP metodu requestu, tedy *get*, *post*, *put* atd. Případně můžeme doplnit *use*, pokud nezáleží na metodě requestu. Za ROUTE

doplníme cestu k requestu, pro což použijeme balíček Path, který si importujeme pomocí request stejně jako express, a za FUNCTION doplníme funkci, která se vykoná, pokud je tento request poslán.Vytvoříme tedy základní request, který nám předá public složku.

```
app.use(express.static(path.join(__dirname, 'public')));
```

Protože jsem použil *use* a nespecifikoval jsem cestu requestu, tento request se zavolá při každém requestu a mám tudíž soubory ve složce public přístupné všude. Funkce express.static je používá pro přenos statických souborů, jako jsou html stránky, css stylesheety a obrázky. ___dirname v Nodu vrací cestu k složce, ve které se program nachází, tedy kořenová složka naší aplikace a funkce path.join spojuje cestu dohromady, což je vhodné, pokud program může běžet na různých systémech s různými oddělovači.

Teď už stačí zavolat *app.listen*, která zajistí, aby server poslouchal na daném portu.

```
app.listen(
 PORT, () => console.log(`Server started on port ${PORT}`)
);
```

3.2.1 Výpočet diskontinuální chromatografie

Kvůli náročnosti výpočtu této simulace chceme, aby probíhal na serverové straně aplikace. Je proto nutné kromě samotné výpočetní funkce vytvořit request, který umožní přijmout všechny potřebné parametry a zaslat výsledek.

Pro výpočet simulace pro diskontinuální chromatografii jsem vytvořil funkci *CalculateDiscon* s těmito parametry:

$Tfeedstart$ - začátek nástřiku [min]
$Tfeedend$ - konec nástřiku [min]
$Tend$ - konec pokusu [min]
$Cfeed$ - koncentrace složky v nástřiku [g/ml]
$EpsT$ - celková mezerovitost kolony [-]
K - adsorpční koeficient složky [-]
L - délka kolony [cm]
Um - rychlost v koloně [cm/min]
k - časový krok diskretizace [min]
h - délkový krok diskretizace [cm]
$useFeedFunc$ - volba uživatele, zda použít pro nástřik definovanou funkci

Tato funkce funguje ve dvou hlavních cyklech. První připraví datovou strukturu. Výsledek výpočtu se dá reprezentovat maticí, kde řádky jsou časové okamžiky a sloupce jsou místa v koloně, tzv. časoprostorové pole koncentrace. Jedna hodnota tedy reprezentuje koncentraci složky v jednom časovém okamžiku na jednom místě v koloně. Tuto matici si v programu definuji jako

2D pole, tedy jedno hlavní pole obsahující velké množství dalších polí, které každé reprezentuje jednu řádku matice, tedy jeden časový okamžik. První cyklus mi tuto datovou strukturu vytvoří a předvyplní počátečními a okrajovými podmínkami.

```
let data = [];
for(let j = 0; j <= Tend/k; j++){
 let tmp = [];
 for(let i = 0; i <= L/h; i++){
  if ( i === 0 && ( j*k >= Tfeedstart ) && ( j*k < Tfeedend ) ){
   tmp.push(Cfeed);
  }
  else{
   tmp.push(0);
  }
 }
data.push(tmp);
}
```

Druhý cyklus je výpočetní. Prochází postupně celou matici a pomocí Wendroffovi formule dopočítává každou hodnotu. Protože se jedná o dopřednou diferenci, je nutné zajistit správný průchod, abychom vždy měli předchozí hodnoty, pomocí kterých dopočítáme hodnoty aktuální. Poté už jen vrátí data.

```
let p = k/h;
let a = Um*(EpsT/(EpsT+((1-EpsT)*K)));
let kof = (1-a*p)/(1+a*p);
for(let j = 0; j < Tend/k; j++){
 for(let i = 0; i < L/h; i++){
  data[j+1][i+1] = data[j][i]+(kof*(data[j][i+1]-data[j+1][i]));
 }
}
return data;
```

3.2.2 Request pro diskontinuální chromatografii

Nyní když máme funkci pro výpočet hotovou, stačí už jen vytvořit request, abychom mohly tyto hodnoty předat. Protože pouze získávám data ze serveru, použiju metodu *get*. Je třeba pouze vymyslet, jak předat všechny argumenty. To můžeme zajistit pomocí cesty následujícím způsobem:

```
app.get(
 '/simDiscon/:Tfs/:Tfe/:Te/:Cf/:EpsT/:K/:L/:Um/:k/:h/:fF',
 ...
)
```

Předávání parametrů zajistíme pomocí cestových parametrů. Když do definice cesty vložíme dvojtečku následovanou nějakým identifikátorem, říkáme expressu, že se jedná o parametr. Za tento parametr může klient doplnit hodnotu proměnné, kterou jsme poté na serverové části získat. Funkce v requestu tedy vypadá takto:

```
(req, res) => {
 let Tfs = Number(req.params.Tfs);
 let Tfe = Number(req.params.Tfe);
 let Te = Number(req.params.Te);
 let Cf = Number(req.params.Cf);
 let EpsT = Number(req.params.EpsT);
 let K = Number(req.params.K);
 let L = Number(req.params.L);
 let Um = Number(req.params.Um);
 let k = Number(req.params.k);
 let h = Number(req.params.h);
 let fF = Number(req.params.fF);
 let data = CalculateDiscon(Tfs,Tfe,Te,Cf,EpsT,K,L,Um,k,h,fF);
 let labels = getLabels(L, h);
 const response = {};
 response.labels = labels;
 response.data = data;
 res.json(response);
}
```

Nejdříve získáme hodnoty z parametrů, zavoláme výpočetní funkci a výsledek uložím do proměnné *data*. Dále si zavolám funkci *getLabels*, která my vrátí pole s popisky pro graf, který budu z těchto dat vykreslovat. Nakonec si vytvořím objekt *response*, do kterého vložím *data* a *label* a odešlu klientovi ve formátu JSON.

3.2.3 Výpočet SMB

Simulace chromatografie v režimu SMB je výrazně složitější. První problém na který narazíme je, že se jedná o kontinuální, a tudíž nekonečný proces. Proto strategie "vypočítej všechno najednou a odešli všechna data zároveň", kterou jsem použil pro diskontinuální chromatografii, není použitelná.

Tento problém jsem vyřešil tak, že jsem výpočetní funkci implementoval jako nekonečný generátor. Funkce mi nyní běží v jednom hlavním cyklu, ve kterém si předvyplní počáteční podmínky a hned vypočítá jeden časový krok, který mi vrátí. To mi také umožnilo změnit datovou strukturu, kde si nyní pro každou kolonu pamatuji pouze předchozí a aktuální časový krok.

Dalším problémem je samotná složitost SMB procesu. Místo jedné kolony máme nyní kolon osm, které na sebe musí navazovat. Také máme další parametry, objemové průtoky čerpadel a časový interval mezi přepnutím čerpadel.

Nejprve vyřešíme rozdílné průtoky. Pro to si systém rozdělíme do čtyř zón, každá zóna jsou dvě kolony mezi čerpadly, tudíž musíme získat čtyři různé průtoky pro každou zónu. Ty získáme z těchto nových parametrů:

Qm - objemový průtok recyklu [ml/min]
Qd - objemový průtok eluentu [mil/min]
Qe - objemový průtok extraktu [mil/min]
Qf - objemový průtok feedu [mil/min]
Qr - objemový průtok rafinátu [mil/min]
diameter - průměr kolony [cm]

Qm je stejný parametr jako Um pro simulaci diskontinuálu, pouze převedena na objemový průtok kvůli jednotnosti. Dále Qe a Qr jsou výstupní průtoky, tudíž budou průtok v zónách za nimi zpomalovat, zatímco Qd a Qf jsou vstupní průtoky a budou jej zrychlovat.

Začneme zónou za Qd, která není závislá na předchozí zóně, pouze na průtoku recyklu a eluentu. Průtok v každé další zóně vypočítáme jako průtok v zóně předchozí, ke kterému přičteme nebo odečteme průtok příslušného ventilu podel toho, jestli je vstupní nebo výstupní. Zároveň si všechny hodnoty převedeme z objemového průtoku na rychlost, protože Wendroffova formule počítá s rychlostí. K tomu potřebujeme průměr kolony.

```
let Um = (Qm / ((Math.PI * diameter)*EpsT));
let Umd = Um + (Qd / ((Math.PI * diameter)*EpsT));
let Ume = Umd - (Qe / ((Math.PI * diameter)*EpsT));
let Umf = Ume + (Qf / ((Math.PI * diameter)*EpsT));
let Umr = Umf - (Qr / ((Math.PI * diameter)*EpsT));
```

Nyní musím zajistit, aby kolony na sebe navazovaly. To dosáhnu ve fázi vyplňování počátečních okrajových podmínek. Pro kolony bez vstupních čerpadel je okrajová podmínka koncentrace na konci předchozí kolony o časový krok zpět. Pro kolony se vstupními ventily musíme zkombinovat vstupující složku a složku z vystupující z předchozí kolony.

```
if (j === 0){
 tmp.push(0);
}
else if ( i === 0 ){
 if ( l === (elPos+2)%8 ){
  tmp.push(data[(l+7)%8][0][data[(l+7)%8][0].length-1]);
  out.extract = data[(l+7)%8][0][data[(l+7)%8][0].length-1];
 }
 else if ( l === (elPos+6)%8 ){
```

```
  tmp.push(data[(l+7)%8][0][data[(l+7)%8][0].length - 1]);
  out.raffinate = data[(l+7)%8][0][data[(l+7)%8][0].length-1];
 }
 else if ( l === elPos ){
  let ratio = Qd / (Umd * ((Math.PI * diameter)*EpsT));
  tmp.push((1-ratio)*data[(l+7)%8][0][data[(l+7)%8][0].length-1]);
 }
 else if ( l === (elPos+4)%8 ){
  let ratio = Qf / (Umf * ((Math.PI * diameter)*EpsT));
  tmp.push(
   (ratio * boundaryFeed) +
   ((1-ratio) * data[(l+7)%8][0][data[(l+7)%8][0].length-1])
  );
 }
 else{
  tmp.push(data[(l+7)%8][0][data[(l+7)%8][0].length-1]);
  }
}
else{
 tmp.push(0);
}
```

První *if* zajišťuje počáteční podmínku, a to že v čase 0 je všude koncentrace 0. *j* značí počet časových kroků. Druhý *if* vyplňuje okrajové podmínky. *i* mi říká, kde na koloně jsem, takže když se rovná nule, jsem na začátku kolony. *l* mi zajišťuje, že jsem na správné koloně. *elPos* mi říká, jaká je pozice čerpadel. Poslední *else* mi dovyplní všude 0 jako dočasnou hodnotu, která bude později přepsána.

Z kódu si můžeme všimnout, že pro výstupová čerpadla si ukládám nějaká data do objektu *out*. Data z výstupových čerpadel jsou hlavním ukazatelem toho, jak proces probíhá a je proto vhodné je mít po ruce. Vytvořil jsem si proto objekt *out*, do kterého si pro každý cyklus uložím hodnoty z výstupních čerpadel a vracím jej spolu s celými daty.

Samotný výpočet pak vypadá takto:

```
if(j !== 0){
 for(let l = 0; l < data.length; l++){
 if( l === elPos || l === (elPos + 1)%8 ){
  newUm = Umd;
 }
 else if( l === (elPos + 2)%8 || l === (elPos + 3)%8 ){
  newUm = Ume;
 }
 else if( l === (elPos + 4)%8 || l === (elPos + 5)%8 ){
```

```
 newUm = Umf;
}
else if( l === (elPos + 6)%8 || l === (elPos + 7)%8 ){
 newUm = Umr;
}
a = newUm*(EpsT/(EpsT+((1-EpsT)*K)));
let kof = (1-a*p)/(1+a*p);
for(let i = 0; i < L/h; i++){
 data[l][1][i+1] =
   data[l][0][i] +
   (kof*(data[l][0][i+1] - data[l][1][i]));
}
}
```

První *if* zajišťuje, že nejsem v časovém kroku nula, kde platí počáteční podmínky a nemám pro něj data, ze kterých bych počítal. Pak začne samotný cyklus, který projde všech osm kolon, podle toho, pro jakou kolonu probíhá výpočet se do proměnné *newUm* přiřadí hodnota rychlosti průtoku pro danou zónu a proběhne výpočet pro každý prostorový krok.

Nyní už jen stačí zajistit rotaci čerpadel. Jak už jsem ukázal dříve, proměnná *elPos* mi definuje polohu čerpadel, protože reprezentuje polohu čerpadla s eluentem a polohy ostatních čerpadel se z ní dají vypočítat. Pro rotaci čerpadel tedy stačí, abychom v časovém intervalu změnili hodnotu *elPos*.

```
cycle = cycle + k;
if(cycle >= switchInterval/60){
 elPos = (elPos + 1)%8;
 cycle = cycle-(switchInterval/60);
}
```

Definoval jsem si proměnou *cycle*, ke které přičítám časový krok. Jakmile proměná přesáhne velikost parametru *switchInterval*, posuneme hodnotu *elPos* o jedna a odečteme interval. *switchInterval* je poslední parametr, který bude uživatel zadávat.

3.2.4 Request pro SMB

Při vytváření requestu narazíme na problém. U diskontinuálu nám stačil jediný request, pomocí kterého jsem předal parametry, spočítal hodnoty a ty jako celek vrátil. Protože jsem ale funkci pro výpočet SMB implementoval jako generátor, toto řešení není možné. První myšlenka je předat uživateli vytvořený generátor. Tu jsem ale rychle zavrhl, protože to jde proti návrhu naší aplikace. Chceme, aby výpočet probíhal na serveru, kdybychom ale předali generátor, musel by se výpočet provádět na straně klienta.

Rozhodl jsem se proto vytvořit requesty dva. První request slouží k tomu, aby uživatel předal parametry serveru. Na serveru se z těchto parametrů vytvoří generátor a unikátní identifikátor, pod kterým se generátor na serveru uloží a který poté vrátí uživateli. Takto tedy vypadá tělo funkce prvního requestu.

```
let id = uuidv4();
let Tfs = Number(req.params.Tfeedstart);
let Tfe = Number(req.params.Tfeedend);
let Te = Number(req.params.Tend);
let Cf = Number(req.params.Cfeed);
let EpsT = Number(req.params.EpsT);
let K = Number(req.params.K);
let L = Number(req.params.L);
let Qm = Number(req.params.Qm);
let k = Number(req.params.k);
let h = Number(req.params.h);
let dia = Number(req.params.diameter);
let Qd = Number(req.params.Qd);
let Qe = Number(req.params.Qe);
let Qf = Number(req.params.Qf);
let Qr = Number(req.params.Qr);
let dT = Number(req.params.deltaT);
let fF = Number(req.params.feedFunc);
SMBconnections[id] = CalculateSMB(Te,Tfs,Tfe,Cf,
 EpsT,K,L,Qm,k,h,dia,Qd,Qe,Qf,Qr,dT,fF);
let response = {
 "id": id
}
res.json(response);
```

Druhý request bude volat klient pokaždé, když chce další časový krok. Protože posílání identifikátorů jako cestové parametry je z mnoha důvodů "bad practice", musíme tento request udělat jako POST a *id* poslat v těle requestu. Díky unikátnímu *id*, se kterým klient tento request volá si pamatujeme, vlastníka generátoru. Funkce druhého requestu tedy vypadá takto:

```
let numberSteps = Number(req.params.numberSteps);
let flag = false;
let id = req.body.uuid;
for(let i = 0; i < numberSteps-1; i++){
let tmpp = SMBconnections[id].next();
let tmp = tmpp.value;
if(!tmp){
```

```
res.json({"value": false});
return;
}
if(tmp[2])
flag = true;
}
let tmpp = SMBconnections[id].next();
let value = tmpp.value;
if(value[2])
flag = true;
let data = [];
for(let i = 0; i < 8; i++){
data.push(value[1][i][1]);
}
let labels = value[0];
const response = {};
response.labels = labels;
response.data = data;
response.rotate = flag;
response.out = value[3];
res.json(response);
```

Opět si získáme parametry, *id* z těla pro identifikaci generátoru a *numberSteps* mi umožňuje přeskakovat několik časových kroků. Klient se díky tomu může rozhodnout přeskakovat časové kroky, což je poté vhodné pro rychlejší vykreslování simulace a zároveň se neztrácí přesnost, která by se ztratila, kdybychom zvětšili časový krok. V proměnné *flag* předávám informaci o tom, jestli došlo k rotaci čerpadel. Dále vytvořím response objekt, který naplním daty a pošlu klientovi ve formátu JSON.

Poslední věc je umožnit klientovy, aby mohl měnit vstupní parametry během výpočtu. Generátory v javascriptu mi umožňují posílat data dovnitř generátorů tak, že je vložíme jako argument do volání *next()*. Uvnitř generátoru potom *yield* vrací data, které jsme dovnitř poslali. Vytvořil jsem si proto proměnou *yieldParam*, do které tyto data vkládám a do cyklu jsem si vložil tuto podmínku:

```
if(yieldParam){
 Umd = Um + (yieldParam[0] / ((Math.PI * diameter)*EpsT));
 Ume = Umd - (yieldParam[1] / ((Math.PI * diameter)*EpsT));
 Umf = Ume + (yieldParam[2] / ((Math.PI * diameter)*EpsT));
 Umr = Umf - (yieldParam[3] / ((Math.PI * diameter)*EpsT));
 switchInterval = yieldParam[4];
 yieldParam = yield "OK";
 j--;
```

```
 continue;
}
```

Když volám *next()* bez argumentu, do *yieldParam* se přiřadí *undefined* a tato část kódu se neprovede. Pokud však volám *next()* s argumentem, přepíšou se vstupní parametry a přeskočím cyklus.

Nyní už stačí vytvořit request, který to umožní klientovi použít. Klient předá *id*, abychom věděli, který generátor klientovi patří, a parametry, na které chce generátor změnit. Funkce pak už je velmi jednoduchá, pouze pro správný generátor zavoláme *next()* s parametry jako argument. Protože potřebujeme *id*, jedná se opět o POST request s *id* v těle.

```
let Qd = Number(req.params.Qd);
let Qe = Number(req.params.Qe);
let Qf = Number(req.params.Qf);
let Qr = Number(req.params.Qr);
let deltaT = Number(req.params.deltaT);
let id = req.body.uuid;
let value = SMBconnections[id].next([Qd, Qe, Qf, Qr, deltaT]).value;
res.json({"value": value});
```

3.3 Klientská část aplikace

V této sekci se budeme zabývat věcmi, které musíme v klientské části aplikace vyřešit, a to vykreslování grafů, Javascriptové funkce a samotné stránky a jejich stylování.

3.3.1 Použití chartjs

Jak již bylo zmíněno, pro grafy použijeme knihovnu **Chartjs**. Tato knihovna používá html prvek *canvas*, do kterého samotný graf vykresluje. Graf potom vytvoříme z kontextu *canvas* prvku a objektu, který definuje druh grafu, data v grafu a jeho nastavení.

```
const ctx = document.getElementById('myChart').getContext('2d');
myChart = new Chart(ctx, {
 type: 'line',
 data: {
  labels: [],
  datasets: []
 },
 options: {
  animation: {
   duration: 0
```

```
  },
  scales: {
   xAxes: [{
    scaleLabel: {
     display: true,
     labelString: 'délková pozice na koloně v cm'
    },
    ticks: {
     autoSkip: true,
     maxTicksLimit: 20
    }
   }],
   yAxes: [{
    scaleLabel: {
     display: true,
     labelString: 'koncentrace v g/ml'
    },
    ticks: {
     beginAtZero: true,
     max: 2,
     min: -1
    }
   }]
  }
 }
});
```

Takto vytvářím prázdný spojnicový graf. Některé vlastnosti které používám jsou *labelString*, která vytvoří popisek k osám, *maxTicksLimit*, která omezuje počet zobrazených popisků os, a *beginAtZero*, která zajistí, že měřítko bude obsahovat nulu. K parametrům tohoto grafu můžeme přistupovat stejně, jako by to byl obyčejný objekt, takže si můžeme nejprve vytvořit prázdný graf a ten si poté naplnit.

Nejprve do grafu vložím prázdné datasety, což jsou objekty definující zobrazení dat do grafu. Pro každou složku naší simulace budeme potřebovat jeden dataset, takže vytvořím tolik datasetů, kolik mám složek. V datasetu se kromě dat definuje popisek a barva, proto jsem si vytvořil funkci, která vytvoří prázdný dataset s vhodnými barvami.

```
function generateDataset(index){
 let dataset = {
  label: '',
  data: [],
  backgroundColor: '',
```

```
  borderColor: '',
  borderWidth: 1,
  pointRadius: 0
 };
 let rand = 'rgba(' +
            colors[index][0] + ', ' +
            colors[index][1] + ', ' +
            colors[index][2];
 let randBackground = rand + ', 0.2)';
 let randColor = rand + ', 1)';
 dataset.backgroundColor = randBackground;
 dataset.borderColor = randColor;
 return dataset;
}
```

Tuto funkci poté použiji v cyklu, který se vykoná tolikrát, kolik mám složek, a každé volání této funkce se jako argument bere index cyklu, což mi zajistí rozdílné barvy.

```
for(let i = 0; i < result.data.length; i++){
 myChart.data.datasets.push(generateDataset(i));
}
```

Data do datasetu pak můžeme vkládat stejným způsobem. Pro aktualizaci grafu na stránce potom na grafu voláme metodu *update()*.

3.3.2 Komunikace se serverem

Komunikace se serverem probíhá pomocí requestů, které jsme si na serveru definovali. Začneme získáním dat pro simulaci diskontinuální chromatografie. Funkce nejdříve získá parametry z formuláře na stránce. Poté vytvoří *url*, pomocí kterého předá veškeré parametry serveru. Poté se zavolá *fetch()* s daným url, což je funkce Javascriptu pro získávání vzdálených zdrojů. Protože odpověď ze serveru je ve formátu JSON, můžeme jí jednoduše převést zpět na Javascriptový objekt. Získané informace předáme do vytvořené datové struktury, kterou poté vrátíme.

```
async function getData(cnt){
 let res = {};
 let data = [];
 let label = [];
 const Tfeedstart = document.getElementById('FS').value;
 const Tfeedend = document.getElementById('FE').value;
 const Tend = document.getElementById('E').value;
 const EpsT = document.getElementById('ET').value;
```

```
 const L = document.getElementById('CL').value;
 const Um = document.getElementById('CS').value;
 k = document.getElementById('TS').value;
 const h = document.getElementById('LS').value;
 for(let i = 1; i <= cnt; i++){
  const Cfeed = document.getElementById('FC' + i).value;
  const K = document.getElementById('AC' + i).value;
  let url = '/simDiscon/' + Tfeedstart + '/' +
            Tfeedend + '/' + Tend + '/' + Cfeed + '/' +
            EpsT + '/' + K + '/' + L + '/' + Um + '/' +
            k + '/' + h + '/' + feedFunc + '/';
  let response = await fetch(url);
  let formres = await response.json();
  data.push(formres.data);
  res.labels = formres.labels;
  res.rotate = formres.rotate;
  label.push('component ' + i);
 }
 res.data = data;
 res.label = label;
 return res;
}
```

Protože request spočítá data pouze pro jednu složku, volám request tolikrát, kolik mám složek. To je zároveň důvod, proč získaná data nevracím rovnou, ale vytvářím nový *res* objekt. Také si můžeme všimnou, že jsem funkci definoval s výrazem *async* a funkci *fetch()* a další funkce volám s výrazem *await*. To mi zajistí, že se tyto funkce volají asynchronně. Díky tomu se mi stránka nezasekne pokaždé, když tyto funkce volám.

Pro získání dat pro SMB vypadá podobně, nejprve získám parametry z formuláře, ze kterých vytvořím url pro request na vytvoření generátoru. Opět použiji *fetch()* a uložím si získané *id*. To vše opět udělám pro každou složku.

```
for(let i = 1; i <= cnt; i++){
 let url = '/simSMB/getConnection/' + Tfeedstart +
           '/' + Tfeedend + '/' + Tend + '/' + Cfeed +
           '/' + EpsT + '/' + K + '/' + L + '/' + Qm +
           '/' + k + '/' + h + '/' + diameter +
           '/' + Qd + '/' + Qe + '/' + Qf + '/' + Qr +
           '/' + switchInterval + '/' + feedFunc + '/';
 let response = await fetch(url);
 let formres = await response.json();
 SMBconnections.push(formres.id);
}
```

Poté vytvořím nekonečný cyklus, který posílá requesty pro výpočet a vrací data. Podobně jako na serverové straně, tuto funkci jsem implementoval jako generátor, takže data získávám postupně voláním *next()*.

```
while(true){
 let res = {};
 let data = [];
 let label = [];
 let out = [];
 for(let i = 0; i < cnt; i++){
  let url = '/simSMB/' + SMBconnections[i] +
            '/' + numberSteps + '/';
  let response = postData(url, {"uuid": SMBconnections[i]});
  if(response.value === false){
   return false;
  }
  data.push(response.data);
  out.push(response.out);
  label.push('component ' + i);
  res.labels = responses.labels;
  res.rotate = response.rotate;
 }
 res.data = data;
 res.label = label;
 res.out = out;
 yield res;
}
```

Protože musím posílat POST request, vytvořil jsem si funkci *postData()*, která mi zavolá *fetch()* s metodou POST a serializací pro JSON.

```
async function postData(url = '', data = {}) {
 const response = await fetch(url, {
  method: 'POST',
  mode: 'cors',
  cache: 'no-cache',
  credentials: 'same-origin',
  headers: {
   'Content-Type': 'application/json'
  },
  redirect: 'follow',
  referrerPolicy: 'no-referrer',
  body: JSON.stringify(data)
 });
```

```
 return response.json();
}
```

Pro změnu vstupních parametrů je už funkce velmi jednoduchá, pouze získám parametry, vytvořím url a zavolám *fetch()*.

```
async function changeParams(){
 const Qd = document.getElementById('VFD').value
 const Qe = document.getElementById('VFE').value
 const Qf = document.getElementById('VFF').value
 const Qr = document.getElementById('VFR').value
 switchInterval = document.getElementById('SI').value;
 for(let i = 0; i < compcnt; i++){
  let url = '/simSMB/change/' + SMBconnections[i] +
            '/' + Qd + '/' + Qe + '/' + Qf + '/' + Qr +
            '/' + switchInterval + '/';
  let response = await postData(url, {"uuid": SMBconnections[i]});
  console.log(response.value);
 }
}
```

3.3.3 Vykreslování dat

Nyní, když data umíme získat, je musíme zobrazit na stránce. Začneme diskontinuálem, protože je jednodušší. Graf už máme vytvořený, takže teď potřebujeme zajistit, aby se pravidelně naplňoval novými daty a vykresloval je. Za tímto účelem použijeme metodu *setInterval()*. Tato metoda přijímá jako parametry funkci a časový interval v milisekundách a přijatou funkci vykoná každý daný interval. Takže potřebujeme vytvořit funkci, která vykoná vše, co tomto intervalu chceme vykonat.

```
getData(compcnt).then(function(result){
 const Tend = document.getElementById('E').value;
 myChart.data.labels = result.labels;
 for(let i = 0; i < result.data.length; i++){
  myChart.data.datasets.push(generateDataset(i));
  myChart.data.datasets[i].label = result.label[i];
 }
 let x = 0;
 intervalHandle = setInterval(function(){
  if( x*k >= Tend )
   clearInterval(intervalHandle);
  for(let i = 0; i < result.data.length; i++){
   myChart.data.datasets[i].data = result.data[i][x];
  }
```

```
  showTime(x*k*60);
  x++;
  myChart.update();
 }, graphspeed);
});
```

Takto data vykreslujeme do grafu, nejprve do grafu vložíme popisky, které jsou neměnné, poté zavolám *setInterval()*. Protože funkce, kterou vkládám do intervalu, je celkem prostá, rozhodl jsem se ji definovat na místě. Funkce pouze naplní graf daty a zavolá na něm metodu *update()*. První *if* zajišťuje, že se interval zastaví, když je simulace u konce. Funkce *showTime()* mi pomáhá zobrazovat čas pokusu.

```
function showTime(timeInSec){
 let sec = Math.round(timeInSec%60);
 let min = Math.floor(timeInSec/60)%60;
 let hour = Math.floor(timeInSec/3600);
 let tstr;
 if( min == 0 && hour == 0 )
  tstr = sec.toFixed(1) + 's';
 else if( hour == 0 )
  tstr = min + 'm ' + sec.toFixed(1) + 's';
 else
  tstr = hour + 'h ' + min + 'm ' + sec.toFixed(1) + 's';
 document.getElementById('Time').innerHTML = tstr;
}
```

Můžeme si všimnout, že na začátku volám funkci *getData()* a na ní volám metodu *then()*. Protože jsme si funkci *getData()* definovali jako asynchronní, všechno co daná funkce vrací je zabalené do promisu. Promise je zvláštní objekt, který mi zajišťuje správný postup operací, i když se provádí asynchronně. To znamená, že funkce, kterou vkládáme do metody *then()*, se vykoná až poté, co získám potřebná data.

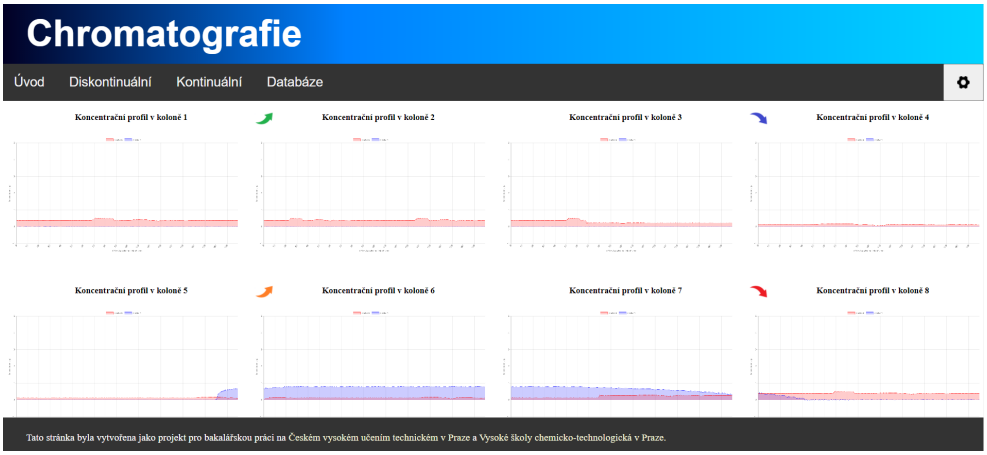
Pro SMB to opět bude o něco složitější. Nejen že zobrazuji data pro osm kolon, zobrazuji ještě čtyři další grafy zobrazující průběh koncentrace jednotlivých složek získaných z extraktu a rafinátu a průběh čistoty za posledních 8 přepnutí. Kdybychom těmto grafům pouze přidávali data, za nějakou dobu by se graf stal nečitelný, zároveň ale nechceme extra data zahazovat, protože mohou být pro uživatele užitečná.

```
gen.next().then((result) => {
 if(!result.value)
  return false;
 if(result.value.rotate === true)
```

```
 rotate();
 for(let x = 0; x < 8; x++){
  myCharts[x].data.labels = result.value.labels;
 }
 labelData.push(result.value.out[0].label);
 for(let x = 8; x < 10; x++){
  let lSlN = labelData.length - ((120/k)/numberSteps);
  myCharts[x].data.labels = labelData.slice(Math.max(lSlN, 1));
 }
 for(let x = 0; x < 10; x++){
  for(let i = 0; i < result.value.data.length; i++){
   if(myCharts[x].data.datasets[i].label === '')
    myCharts[x].data.datasets[i].label = result.value.label[i];
  }
 }
 for(let j = 0; j < 8; j++){
  for(let i = 0; i < result.value.data.length; i++){
   myCharts[j].data.datasets[i].data = result.value.data[i][j];
  }
  myCharts[j].update();
 }
 for(let i = 0; i < result.value.out.length; i++){
  extrData[i].push(result.value.out[i].extract);
  raffData[i].push(result.value.out[i].raffinate);
  let eSlN = extrData[i].length-((120/k)/numberSteps);
  let rSlN = raffData[i].length-((120/k)/numberSteps);
  let eSl = extrData[i].slice(Math.max(eSlN, 1));
  let rSl = raffData[i].slice(Math.max(rSlN, 1));
  myCharts[8].data.datasets[i].data = eSl;
  myCharts[9].data.datasets[i].data = rSl;
 }
 myCharts[8].update();
 myCharts[9].update();
 showTime(result.value.out[0]);
 let sw = document.getElementById('Switch');
 sw.innerHTML = switches.toString();
}).then((result) => {
 if(!stopflag){
 plotNextStep(gen);
 }
});
```

Funkce se tedy skládá z několika cyklů, které prochází všechny grafy a plní je správnými daty. Pro grafy rafinátu a extraktu si všimneme, že nejprve

data vkládáme do *labelData*, *extrData* a *raffData*, na které potom voláme *slice*. Tím se zajistí, že do grafu zobrazíme pouze nejnovější část dat. Dále si všimneme, že pokud z dat získáme informaci o tom, že došlo k přepnutí čerpadel, zavoláme metodu *rotate()*, která přepnutí graficky zobrazí. Zároveň inkrementuje hodnotu *switches*, kterou používáme k zobrazení počtu přepnutí čerpadel.

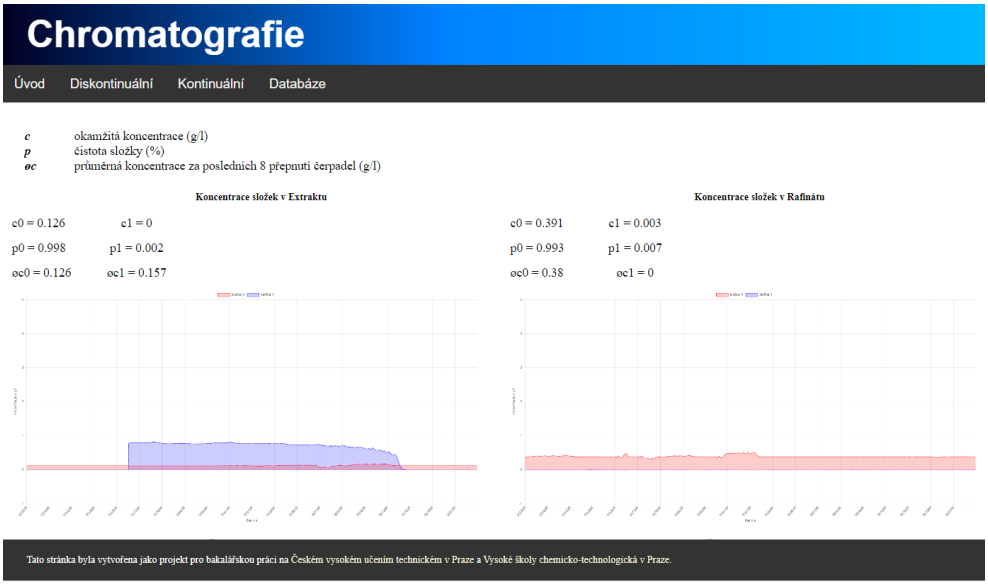


Obrázek 3.1: Ukázka stránky s osmy grafy reprezentující stav osmy kolon

Také si můžete všimnou konce funkce, kde tuto funkci volám rekurzivně. To protože tuto funkci nevolám v *setInterval()*. Protože *setInterval()* funguje takovým způsobem, že když volá funkci rychleji, než se stihne vykonat, vloží ji do fronty a vykoná se až poté co se vykonají všechny funkce před ní. To způsobovalo několik problémů, především, když danou simulaci chceme pozastavit. Nahradil jsem tedy *setInterval()* koncovou rekurzí a pozastavení simulace řídím pomocí proměnné *stopflag*.

Grafy pro zobrazení čistoty vykresluji v úplně jiném intervalu. Protože průměruji čistotu za posledních osm přepnutí, dává smysl přidat hodnotu po každém přepnutí. Zároveň hodnoty počítám z průměrné koncentrace za posledních osm přepnutí. Vytvořil jsem si proto zvlášť funkci pro spočítání a vykreslení čistot pro rafinát a extrakt, kterou poté volám z funkce *showAvgConcentration()*, která se mi stará o zobrazení průměrných koncentrací a je volána každou rotaci čerpadel.

```
function graphAvgPurity(extrArr, raffArr){
 let extrSum = extrArr.reduce(function(a, b){
  return a + b;
 }, 0);
 let raffSum = raffArr.reduce(function(a, b){
  return a + b;
 }, 0);
```



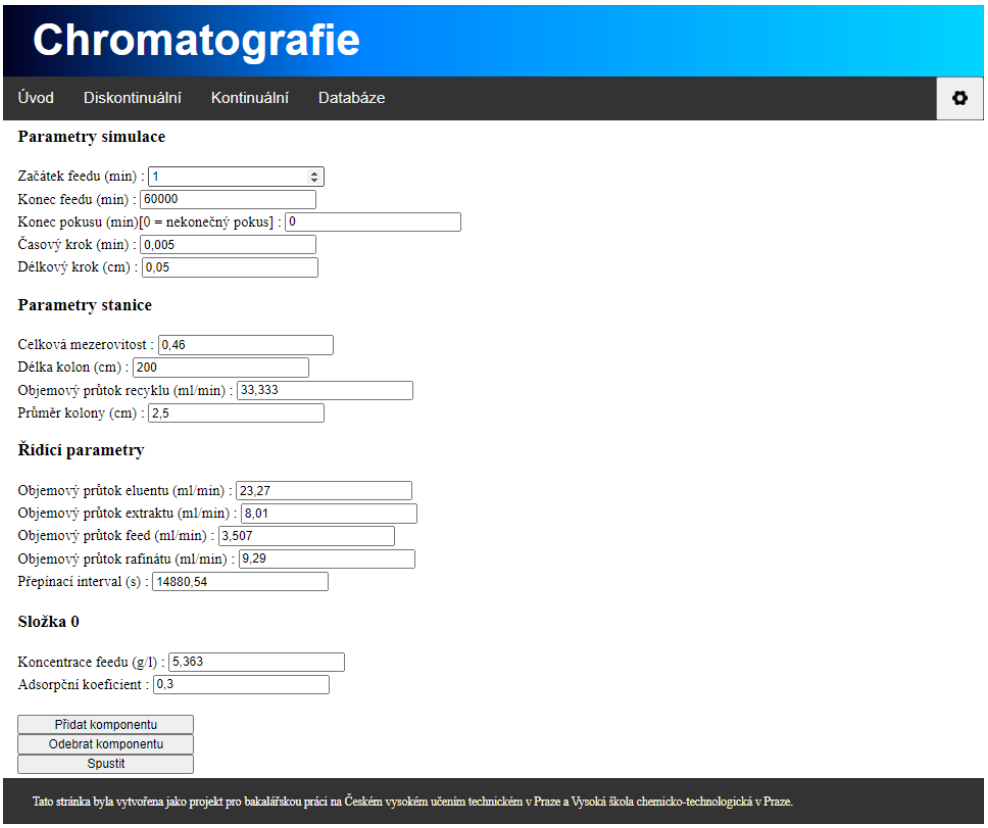
Obrázek 3.2: Ukázka stránky s grafy pro rafinát a extrakt

```
myCharts[10].data.labels.push(
 myCharts[8].data.labels[myCharts[8].data.labels.length-1]
);
myCharts[11].data.labels.push(
 myCharts[9].data.labels[myCharts[9].data.labels.length-1]
);
for(let i = 0; i < compcnt; i++){
if(extrSum == 0)
 myCharts[10].data.datasets[i].data.push(0);
else{
 let tmp = 100*extrArr[i]/extrSum;
 myCharts[10].data.datasets[i].data.push(tmp);
 }
if(raffSum == 0)
 myCharts[11].data.datasets[i].data.push(0);
else{
 let tmp = 100*raffArr[i]/raffSum
 myCharts[11].data.datasets[i].data.push(tmp);
 }
}
myCharts[10].update();
myCharts[11].update();
}
```

3.3.4 Stránky a stylování

Jedná se o standardní HTML stránky. Každá stránka má záhlaví s navigační lištou a zápatí. Stránky pro simulace obsahují formulář pro zadání parametrů a část, kde se získaná data zobrazují. Také obsahují vyskakovací okénko nastavení, ve kterém se dají nastavit věci jako zredukování počtu vykreslených bodů v grafu a rychlost vykreslování.

Styluji pomocí CSS. Pro rozložení stránky používám především zobrazení *grind*, zobrazení *flex* poté používám pro rozložení jednotlivých elementů v sekcích stránky.



Obrázek 3.3: Ukázka stránky s formulářem

Aby simulace vypadala živě, vytvořil jsem si Javascriptové funkce, které mi mění některé elementy a styly stránky za běhu simulace. Pro zobrazení posunu čerpadel používám již zmíněnou funkci *rotate*.

```
function rotate(){
 switches++;
 let p = [];
 for(let i = 0; i < 8; i++){
```

```
  p[i]=document
        .querySelector(".myChart"+(i+1)+"point")
        .innerHTML
 }
 for(let i = 0; i < 8; i++){
  document
    .querySelector(".myChart"+(i+1)+"point")
    .innerHTML
    = p[(i+7)%8]
 }
 showAvgConcentration();
}
```

Tato funkce posouvá obsah elementů, které obsahují indikátor toho, jaké čerpadlo je zrovna připojeno k dané koloně. Zároveň mi volá funkci *showAvgConcentration()*, která mi počítá a zobrazuje průměrnou koncentraci na výstupních čerpadlech za posledních osm cyklů tak, že vezme konec pole hodnot koncentrací odpovídající posledním osmi cyklům, spočítá průměr a napíše ho dovnitř elementu. Tato hodnota se mění při každém přepnutí, proto jí volá tato funkce.

```
function showAvgConcentration(){
 const arrAvg = arr=>arr.reduce((a,b)=>a+b,0)/arr.length;
 for(let i = 0; i < compcnt; i++){
  document.querySelector(".myChart9avgconc" + (i+1))
   .innerHTML = "øc" + i + " = " +
   (Math.round(arrAvg(extrData[i].slice(Math.max(
    extrData[i].length-(((8*switchInterval/60)/k)/numberSteps),0
   ))) * 1000) / 1000);
  document.querySelector(".myChart10avgconc" + (i+1))
   .innerHTML = "øc" + i + " = " +
   (Math.round(arrAvg(raffData[i].slice(Math.max(
    raffData[i].length-(((8*switchInterval/60)/k)/numberSteps),0
   ))) * 1000) / 1000);
 }
}
```

Další funkce pro zobrazení dat na stránce, jsou *showConcentration()* a *showPurity()*, které mi pouze zobrazí aktuální hodnoty koncentrace a čistoty na rafinátu a extraktu.

```
function showConcentration(){
 for(let i = 0; i < compcnt; i++){
  document.querySelector(".myChart9conc" + (i+1)).innerHTML =
```

```
   "c" + i + " = " + (Math.round(
    extrData[i][extrData[i].length-1] * 1000
   ) / 1000);
  document.querySelector(".myChart10conc" + (i+1)).innerHTML =
   "c" + i + " = " + (Math.round(
    raffData[i][raffData[i].length-1] * 1000
   ) / 1000);
 }
}

function showPurity(){
 let totalConcExt = 0;
 let totalConcRaff = 0;
 for(let i = 0; i < compcnt; i++){
  totalConcExt = totalConcExt +
   extrData[i][extrData[i].length-1];
  totalConcRaff = totalConcRaff +
   raffData[i][raffData[i].length-1];
 }
 for(let i = 0; i < compcnt; i++){
  let tmpExt = (Math.round(
   (extrData[i][extrData[i].length-1]/totalConcExt) * 1000
  ) / 1000);
  let tmpRaf = (Math.round(
   (raffData[i][raffData[i].length-1]/totalConcRaff) * 1000
  ) / 1000);
  if(isNaN(tmpExt) || isNaN(tmpRaf))
   return;
  document.querySelector(".myChart9pur" + (i+1)).innerHTML =
   "p" + i + " = " + tmpExt;
  document.querySelector(".myChart10pur" + (i+1)).innerHTML =
   "p" + i + " = " + tmpRaf;
 }
}
```

Pro přidávání a odebírání složek jsem vytvořil dvě funkce, které mi přidávají a odebírají složky ze stránky, takže se zobrazuje vždy jen vybraný počet složek. Tyto funkce mi zároveň upraví hodnoty *compcnt*, což je reprezentace počtu složek v programu, a *graphspeed*, což ovlivňuje délku intervalu pro funkci vykreslení grafu. Čím více složek, tím delší je vykreslení, proto je třeba upravit délku intervalu.

```
function addComponent(){
 if (compcnt >= 5)
```

```
  return;
 let tmpid = 'comp' + compcnt;
 compcnt++;
 graphspeed = 60 + (60 * compcnt);
 let tmp = document.getElementById(tmpid);
 let tmphtml = '<div class="comp" id="comp' + compcnt +
  '"><h3>Složka ' + (compcnt-1) +
  '</h3><label>Koncentrace feedu (g/l) :' +
  ' <input type = "number" name="feed_concentration" id="FC' +
  compcnt +
  '" value="3.4" /></label><label>Adsorpční koeficient : ' +
  '<input type = "number" name="adsorbtion_coeficient" id="AC' +
  compcnt +
  '" value="12.089" /></label></div>';
tmp.insertAdjacentHTML('afterend', tmphtml);
}

function removeComponent(){
if ( compcnt <= 1 )
return;
let tmpid = 'comp' + compcnt;
let tmp = document.getElementById(tmpid);
tmp.remove();
compcnt--;
graphspeed = 60 + (60 * compcnt);
}
```

Dalším problémem byl vzhled stránky před spuštěním simulace. Protože grafy vytvořím až při spuštění simulace, před spuštěním na místech kde budou grafy je prázdné místo, což nevypadá dobře, když už jsou pro grafy zobrazeny nadpisy a další informace. Proto jsem nadpisy na výchozí stránce nevyplnil a některé další prvky schoval a vytvořil jsem funkci, která všechny tyto informace vyplní. Tuto funkci poté volám jednou při spuštění simulace.

```
function setDefaultRotation(){
 document.querySelector(".processInfo").style.display = "grid";
 document.querySelector(".hide").style.display = "block";
 document.querySelector(".myChart1point").innerHTML =
  '<img src="images/feedarrow.png"
  alt="feed" width="24" height="16">';
 document.querySelector(".myChart3point").innerHTML =
  '<img src="images/raffarrow.png"
  alt="raff" width="24" height="16">';
 document.querySelector(".myChart5point").innerHTML =
```

```
  '<img src="images/eluentarrow.png"
  alt="elue" width="24" height="16">';
 document.querySelector(".myChart7point").innerHTML =
  '<img src="images/extractarrow.png"
  alt="extr" width="24" height="16">';
 document.querySelector(".myChart1number").innerHTML =
  "Koncentrační profil v koloně 1";
 document.querySelector(".myChart2number").innerHTML =
  "Koncentrační profil v koloně 2";
 document.querySelector(".myChart3number").innerHTML =
  "Koncentrační profil v koloně 3";
 document.querySelector(".myChart4number").innerHTML =
  "Koncentrační profil v koloně 4";
 document.querySelector(".myChart5number").innerHTML =
  "Koncentrační profil v koloně 5";
 document.querySelector(".myChart6number").innerHTML =
  "Koncentrační profil v koloně 6";
 document.querySelector(".myChart7number").innerHTML =
  "Koncentrační profil v koloně 7";
 document.querySelector(".myChart8number").innerHTML =
  "Koncentrační profil v koloně 8";
 document.querySelector(".myChart9desc").innerHTML =
  "Koncentrace složek v Extraktu";
 document.querySelector(".myChart10desc").innerHTML =
  "Koncentrace složek v Rafinátu";
 document.querySelector(".myChart11desc").innerHTML =
  "Čistota složek v Rafinátu";
 document.querySelector(".myChart12desc").innerHTML =
  "Čistota složek v Extraktu";
}
```

Poslední vlastností stránky je stažení koncentračního profilu z výstupních čerpadel. Tyto profily jsou nejdůležitější ukazatele průběhu chromatografického procesu, je proto nutné umět tuto informaci ze stránky získat pro další použití. Vytvořil jsem si proto dvě funkce, které mi umožní stáhnout tato data ve formátu CSV. První funkce stáhne tyto data pro poslední dvě hodiny simulace, druhá mi stáhne veškerá data z průběhu simulace.

```
function getCSVlast2h(){
 let csv = "";
 let tmpRaffData = [];
 let tmpExtrData = [];
 let tmpLabelData = labelData.slice(
  Math.max(labelData.length - ((20/k)/numberSteps), 1)
```

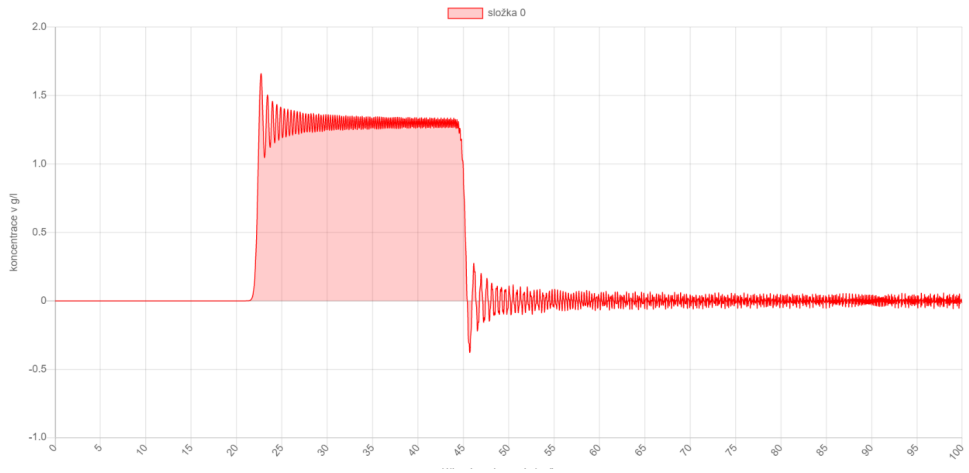
```
 );
 for(let j = 0; j < compcnt; j++){
  tmpRaffData.push(raffData[j].slice(
   Math.max(raffData[j].length - ((20/k)/numberSteps), 1))
  );
  tmpExtrData.push(extrData[j].slice(
   Math.max(extrData[j].length - ((20/k)/numberSteps), 1))
  );
  csv = csv + "raff" + (j+1) + ",extr" + (j+1) + ",";
 }
 csv = csv + "timeS\r\n";
 for(let i = 0; i < tmpLabelData.length; i++){
  for(let j = 0; j < compcnt; j++){
   csv = csv + tmpRaffData[j][i] + "," + tmpExtrData[j][i] + ",";
  }
  csv = csv + tmpLabelData[i] + "\r\n";
 }
 let element = document.createElement('a');
 element.href = window.URL.createObjectURL(
  new Blob([csv], {type: 'text/csv'})
 );
 element.download = 'last2h.csv';
 element.style.display = 'none';
 document.body.appendChild(element);
 element.click();
 document.body.removeChild(element);
}

function getCSVfull(){
 let csv = "";
 for(let j = 0; j < compcnt; j++){
  csv = csv + "raff" + (j+1) + ",extr" + (j+1) + ",";
 }
 csv = csv + "timeS\r\n";
 for(let i = 0; i < labelData.length; i++){
  for(let j = 0; j < compcnt; j++){
   csv = csv + raffData[j][i] + "," + extrData[j][i] + ",";
  }
  csv = csv + labelData[i] + "\r\n";
 }
 let element = document.createElement('a');
 element.href = window.URL.createObjectURL(
  new Blob([csv], {type: 'text/csv'})
 );
```

```
 element.download = 'fullData.csv';
 element.style.display = 'none';
 document.body.appendChild(element);
 element.click();
 document.body.removeChild(element);
}
```

Závěr

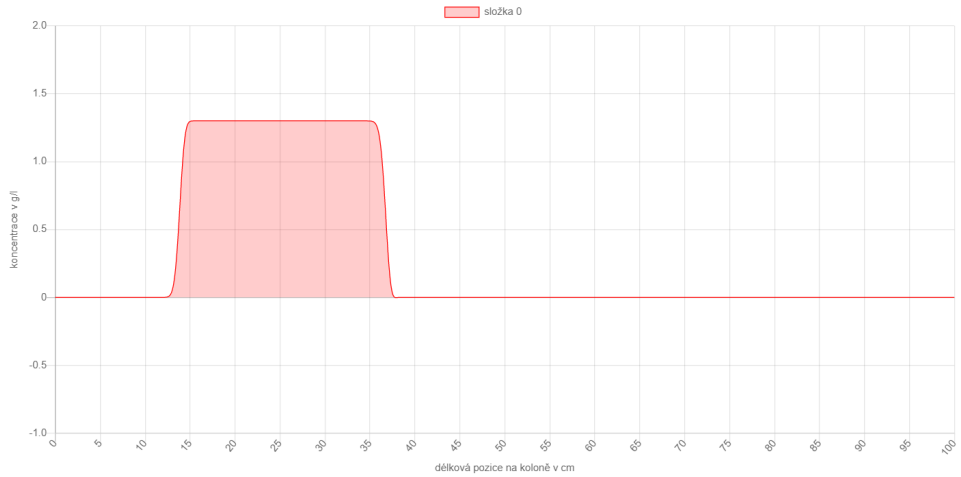
Samotná matematická simulace reálného chromatografického procesu vykazovala charakteristické vlastnosti systému. Na koncentračních křivkách byl pozorován numerický šum při skokové změně vstupní koncentrace. Tento šum je pro danou formuli typický.

Pro eliminaci tohoto šumu byla použita spojitá náběhová funkce. Z důvodu poměrné složitosti výpočtu v kontinuálním režimu byla tato metoda použita pouze u diskontinuálních simulací. Jelikož je Wenderoffova formule za daných podmínek vždy konvergentní, není přítomnost numerického šumu zásadní.



Obrázek 3.4: Ukázka grafu bez použití náběhové funkce

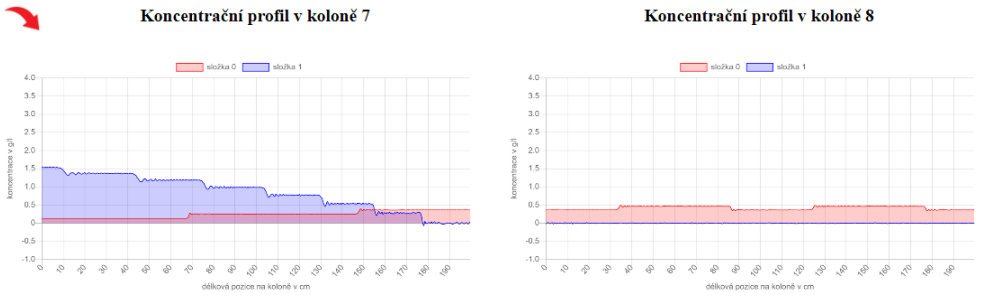
Žádoucím výsledným stavem systému SMB chromatografie jsou čisté výstupní proudy, to znamená, že v extraktu i v rafinátu je zastoupena pouze jedna separovaná složka. Tohoto rovnovážného stavu je však poměrně těžké dosáhnout, jelikož je nutné přesné nastavení operačních parametrů. Velmi malá nepřesnost může způsobit znečištění výstupních proudů.



Obrázek 3.5: Ukázka grafu s použitou náběhovou funkcí

Obrázek 3.6: Ukázka grafu náběhu separace složky 0 v rafinátu

Z programátorské stránky jsem se potýkal se standardními problémy pro tento druh aplikace, komunikace mezi serverem a klientem, serializace a zpracování dat. Poměrně problematické bylo vizuální rozložení stránky, tak aby bylo uživatelsky přehledné. U kontinuální SMB chromatografie vykresluji velké množství dat a bylo náročné je vměstnat na jednu stránku v rozumném rozpoložení. Jak už bylo zmíněno, použil jsem zobrazení grid, které se osvědčilo, přináší ovšem několik problémů. Pro každý graf jsem musel vyčlenit určité množství místa. Když jsem se snažil o relativní velikosti, docházelo k deformaci grafů. Rozhodl jsem se proto pro absolutní jednotky, což sice zamezí deformaci grafů, muže to ale způsobovat nevhodné zobrazení pro různé veli-



Obrázek 3.7: Ukázka grafu koncentrace na kolonách 7 a 8, kde můžeme pozorovat separaci složek

kosti obrazovek.

U klientské části aplikace se ukázalo, že rychlé vykreslování grafů je příliš výkonnostně náročné a způsobuje zpomalení běhu aplikace. Abych zpomalení zamezil, vytvořil jsem funkci *trim()*, která mi umožňuje periodicky vynechávat hodnoty z pole dat, čímž zmenšuji počet vykreslených bodů v grafu.

Analogicky jsem vytvořil i možnost periodicky vynechávat časové kroky, kterou jsem zabudoval přímo do requestu, který volám pro získání dalšího časového kroku simulace. Tímto způsobem mohu značně zrychlit zobrazení simulovaných dat, aniž bych přišel o jejich přesnost.

Abych zajistil asynchronitu aplikace, použil jsem kombinaci koncové rekurze v promisu a funkce *setInterval()*. Koncovou rekurzi jsem použil pro opakované volání vykreslovací funkce, kterou volám velmi často a proto *setInterval()* není vhodný. *setInterval()* jsem poté použil u funkcí, které zobrazují okamžitou koncentraci a čistotu, které volám v delších intervalech a nemohou tudíž zahltit frontu událostí.

Literatura

[1] Zhang, Y.; Feng, L.; Seidel-Morgenstern, A.; aj.: Accelerating optimization and uncertainty quantification of nonlinear SMB chromatography using reduced-order models. *Computers & Chemical Engineering*, ročník 96, 09 2016, doi:10.1016/j.compchemeng.2016.09.017.

[2] eMarketer. (n.d.): Most used libraries, frameworks, and tools among developers, worldwide, as of early 2020. 2020. Dostupné z: `https://www.statista.com/statistics/793840/worldwide-developer-survey-most-used-frameworks/`

[3] Schoenmakers, P.: Chromatography in Industry. *Annual Review of Analytical Chemistry*, ročník 2, č. 1, 2009: s. 333–357, doi:10.1146/annurev-anchem-060908-155133, pMID: 20636066, `https://doi.org/10.1146/annurev-anchem-060908-155133`. Dostupné z: `https://doi.org/10.1146/annurev-anchem-060908-155133`

[4] Rajendran, A.; Paredes, G.; Mazzotti, M.: Simulated moving bed chromatography for the separation of enantiomers. *Journal of Chromatography A*, ročník 1216, č. 4, 2009: s. 709 – 738, ISSN 0021-9673, doi:https://doi.org/10.1016/j.chroma.2008.10.075, editors' Choice III. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0021967308018451`

[5] Vogel, H.; Todaro, C.: *Fermentation and Biochemical Engineering Handbook, 2nd Ed.: Principles, Process Design and Equipment*. Elsevier Science, 1996, ISBN 9780815517139. Dostupné z: `https://books.google.cz/books?id=qBfk8keNDbAC`

[6] Rodrigues, A.: *Simulated Moving Bed Technology: Principles, Design and Process Applications*. Elsevier Science, 2015, ISBN 9780128020517. Dostupné z: `https://books.google.cz/books?id=qPGcBAAAQBAJ`

[7] Bellot, J.; Condoret, J.: Modelling of liquid chromatography equilibria. *Process Biochemistry*, ročník 28, č. 6, 1993: s. 365 – 376, ISSN 1359-5113, doi:https://doi.org/10.1016/0032-9592(93)80023-A. Dostupné z: http://www.sciencedirect.com/science/article/pii/003295929380023A

[8] DeVault, D.: The Theory of Chromatography. *Journal of the American Chemical Society*, ročník 65, č. 4, 1943: s. 532–540, doi:10.1021/ja01244a011, https://doi.org/10.1021/ja01244a011. Dostupné z: https://doi.org/10.1021/ja01244a011

[9] Zhong, G.; Guiochon, G.: Simulated moving bed chromatography. Effects of axial dispersion and mass transfer under linear conditions. *Chemical Engineering Science*, ročník 52, č. 18, 1997: s. 3117 – 3132, ISSN 0009-2509, doi:https://doi.org/10.1016/S0009-2509(97)00133-4. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0009250997001334

[10] Morton, K.; Mayers, D.: *Numerical Solution of Partial Differential Equations: An Introduction*. Cambridge University Press, 2005, ISBN 9781139443203. Dostupné z: https://books.google.cz/books?id=GW6_AwAAQBAJ

[11] Thomas, J.: *Numerical Partial Differential Equations: Finite Difference Methods*. Texts in Applied Mathematics, Springer New York, 2013, ISBN 9781489972781. Dostupné z: https://books.google.cz/books?id=83v1BwAAQBAJ

[12] Lax, P.; Wendroff, B.: Systems of Conservation Laws. 1959.

[13] Gupta, H.: Top 10 Frameworks for Web Applications. 2020. Dostupné z: https://www.geeksforgeeks.org/top-10-frameworks-for-web-applications/

[14] INDIA, S.: Why Node Js is Very Popular Among Fortune 500 Companies ? 2019. Dostupné z: https://medium.com/quick-code/node-js-and-fortune-500-companies-fewer-efforts-more-rewards-282db19160c0

[15] de Moor, T.: MOST POPULAR NODE.JS FRAMEWORKS IN 2019. 2019. Dostupné z: https://x-team.com/blog/most-popular-node-frameworks/

Příloha A

Seznam symbolů

c	Koncentrace, g/l
q^*	Rovnovážná koncentrace v sorbentu, g/l
K	Adsorpční koeficient
Q_{max}	Kapacita sorbentu, g/l
u_m	Rychlost toku, cm/min
ε	Celková porozita sorbentu
D_ax	Axiální disperzní koeficient, cm^2/min
t	Čas, min
x	Axiální koordináta, cm
ς	Časový rozptyl přechodové funkce, min

Příloha B

Seznam použitých zkratek

SMB Simulated moving bed

JSON Javascript object notation

NPM Node package manager

HTML Hypertext markup language

CSS Cascading style sheets

HTTP Hypertext transfer protocol

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD.
- `src` ..... zdrojové soubory práce
  - `impl` ..... zdrojové kódy implementace
    - `public` ..... stránky a zdroje klienta
      - `javascript` ..... zdrojové kódy javascriptových funkcí klienta
      - `images` ..... obrázky použité v klientovi
      - `CSS` ..... kaskádové styly použité v klientovi
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS