

## Ambiente virtual de aprendizagem

Objetivo: construção de um ambiente virtual de aprendizagem (AVA) para o ensino de modelagem computacional.

### Requisitos da aplicação

- Deve ser fácil criar um novo tipo de modelo.
- Deve ser fácil definir o tipo de visualização para o modelo.
- Deve ser fácil definir as características (tipo, tamanho, posição do gráfico, controles, posição dos controles, etc) de uma visualização e exibir essas características na view.
- Deve ser fácil criar um novo problema e fazer a associação dele com um ou mais modelos (para fins de simulação).
- Deve ser fácil carregar a descrição de um problema na view.
- Deve ser fácil para o usuário navegar entre os vários problemas.
- Deve ser fácil modificar quaisquer características de um problema, modelo ou visualização.

### Arquitetura

Conceitos principais: assunto e modelo.

Um assunto é um conteúdo sobre qualquer assunto que será apresentado na aplicação. Um modelo possui um assunto (que dá contexto, motivação para o modelo) e informações a mais sobre o modelo. Um assunto pode estar ligado a zero ou mais modelos relacionados àquele assunto.

Módulos GuiManager, Subject, Model, Visualization.

Módulo GuiManager: Gerencia todas as telas e os eventos em cada uma delas. Processa as requisições e as envia para os módulos Problem e Model.

Classes presentes nesse módulo: ControllerProblem, ControllerModel, ControllerVisualization.

Módulo Subject: Gerencia todo o conteúdo dos problemas disponíveis. Possui uma ligação com o módulo Visualization para definir características da visualização. Possui uma ligação com o Módulo Model para definir quais modelos modelam o problema. A descrição de um problema será armazenada no seguinte formato (que será convertido para Json usando Json::Value):

```
uid: Número identificador único;  
title: ...;  
introduction: Texto formatado da introdução;  
maintext: Texto principal;  
images: Lista de paths (path1, path2, ..., pathn);  
models: Lista de modelos  
results: [imagem: path, descricao: ...; imagem: path2, descricao: ...];
```

Módulo Model: Gerencia todo o conteúdo dos modelos disponíveis. Possui uma ligação com o módulo Visualization para definir características da visualização.

A descrição do modelo seguirá o seguinte formato (que será convertido para Json):

uid: Número identificador único;  
title: ...;  
maintext: Texto principal;  
images: Lista de paths (path1, path2, ..., pathn);  
codes: Lista de codigos (path1, path2, ..., pathn);  
results: [imagem: path, descricao: ...; imagem: path2, descricao: ...];

Obs: Os parágrafos de introduction e do maintext podem ser divididos com um array Json. Estudar outras formas de estruturar esse conteúdo sem ficar utilizando tags ou outras marcações. Tentar não utilizar marcação ou utilizar o mínimo possível (Separar o conteúdo da apresentação dele).

ODEModel:  
equation: string (A equação será compilada)

CAModel:  
scripts: Lista de scripts

Módulo Visualization: Responsável por gerenciar toda a parte de visualização dos problemas e modelos. Requisitos: organizar a informação em containers (com propósitos específicos) de forma a facilitar a atualização da interface.

## Classes

Model, CAModel, ODEModel, PDEModel, MatrixModel, Subject, Service, Visualization.

A classe CAModel facilitará a criação de vários tipos de autômatos celulares suportando autômatos com vários tipos de populações e com vários estados. Facilitará também a definição de regras de transição, condição inicial e outras características através de scripts escritos em Lua. Lua foi escolhida dada a sintaxe simplificada, eficiência e fácil integração com C.

Interfaces para a entrada de uma **regra de transição**:

1)

|                 |              |        |
|-----------------|--------------|--------|
| tipo_de_celula  | estado_atual |        |
| estado_vizinhos | num_vizinhos | op_cmp |

Dropdown list para todos os campos.

2) Script Lua

As classes ODEModel e PDEModel irão trabalhar com um compilador de EDOs e EDPs. Será definida uma gramática para especificar a sintaxe das EDOs e EDPs. O usuário do ambiente poderá digitar as EDOs e EDPs em um formato amigável. Essa entrada será processada pelo compilador que irá gerar códigos em C e Cuda que implementam computacionalmente a equação. Se todos os passos forem bem sucedidos, o código gerado será executado e os resultados serão exibidos na interface.

A informação sobre se o ID é um parâmetro ou variável é passada no início do arquivo de entrada. Por exemplo, para o modelo logístico a entrada seria:

```
time: inicio to fim
vars: n
params: r, k, m
ini
n = 2;
r=0.1;
k=100;
m=0.05;
;
n = r*n*(1 - n/k) - m*n ;
```

O código gerado será injetado em pontos específicos de um arquivo template (com várias partes em comum já implementadas):

```
//begin_ini seção de inicialização de variáveis //end_ini
//begin_param seção de inicialização dos parâmetros //end_param
//begin_ode seção que calcula as ODEs //end_ode
//begin_savefile seção que salva os resultados em arquivos //end_savefile
//begin_plot seção que gera os gráficos dos resultados //end_plot
```

Durante a análise sintática, também será construído um grafo que mostra a relação entre as variáveis. Criação de uma Relation que representará um arco no grafo.

#### Gramática LR:

```
S' -> S
S -> vars ListaNomes params ListaNomes ini ListaEq ; ListaEq
ListaNomes -> id
ListaNomes -> id , ListaNomes
ListaEq -> Eq ListaEq
ListaEq -> Eq
Eq -> id = Unary ListaT ;
Eq -> id = ListaT ;
ListaT -> T ListaT
ListaT -> T
T -> T Binary T
T -> Operando
Binary -> +
Binary -> -
Binary -> *
Binary -> /
Operando -> id
Operando -> num
Operando -> ( T )
Unary -> +
Unary -> -
```

```
vars id , id params id , id , id ini id = num ; ; id = - id ;
```

- Todas as variáveis e parâmetros serão inseridos na tabela de símbolos.
- Criar listas para a tabela LR(0).

## Classe Model

Define os atributos e funcionalidades em comum de todos os modelos.

## Classe ResourceProvider (provedor de recursos)

1) Mantém um vetor com todos os problemas e um vetor com todos os modelos:  
`vector<const &Problem> problems;`  
`vector<const &Model> models;`

2) Criação de **hashmaps**:  
`map<"nome_do_problema",Problem*> problems;`  
`map<"nome_do_modelo",Model*> models;`

Sempre que uma requisição for feita, será enviado o nome do problema/modelo (que pode ser o mesmo nome do arquivo txt com suas informações) na url. O controlador processa a requisição e realiza as ações necessárias para retornar a view do problema/modelo. Podemos recuperar rapidamente um problema ou modelo que já foi carregado na memória consultando os maps da classe ResourceProvider que atua como uma cache nesse caso.

No caso da simulação interativa, é importante estabelecer uma sessão com o cliente ou até mesmo criar uma conexão web socket por questões de performance.

## Classe Visualization

A classe Visualization será responsável por processar o Json e definir o conteúdo html da resposta. Ela será chamada pela classe Controladora que irá passar o Json para Visualization e receber o html de volta. O html será inserido no body de uma resposta http normal criada com `HttpResponse::newHttpResponse()`. A classe Visualization trabalha com o conteúdo do modelo e com as informações sobre o tipo de visualização para o modelo em questão.

Processamento do Json (comparar a eficiência no lado servidor x lado cliente):

Considerando o array de uma section:

- . Se o valor for string, será criado um novo parágrafo com `<p>`.
- . Se o valor for image, será criada uma imagem com `<img>`.

## Passos futuros

- Fazer diagramas de sequência para modelar as requisições.
- Desenvolver scripts de teste com curl e implementar casos de testes.
- Criação de um parser que processa o texto de um assunto e gera o Json:

\$\$

Quando estamos estudando modelos, é útil identificar categorias amplas. A classificação dos modelos nessas categorias vai nos dizer algumas coisas sobre a estrutura deles.

\$Estágios da modelagem\$

...

\$\$: marca o início de uma nova sessão (que pode ter um título ou não).

\n: Uma linha em branco marca a abertura de um novo parágrafo.

### Linguagens, frameworks e outras tecnologias

C, C++, Lua, Cuda, HTML, CSS, Javascript, JQuery, Bootstrap (containers, cards, badges), Drogon, Jsoncpp.

2d plot with Javascript: `plotly.js`

### Princípios de projeto

Alta coesão, baixo acoplamento, especialista na informação, information hiding, etc.

### Gramática LL

$S' \rightarrow S$

$S \rightarrow \text{vars ListaNomes params ListaNomes ini ListaEq ; ListaEq}$

$\text{ListaNomes} \rightarrow \text{id ListaNomes2}$

$\text{ListaNomes2} \rightarrow , \text{id ListaNomes2} \mid \epsilon$

$\text{ListaEq} \rightarrow \text{Eq ListaEq2}$

$\text{ListaEq2} \rightarrow \text{Eq ListaEq2} \mid \epsilon$

$\text{Eq} \rightarrow \text{id} = \text{Eq2}$

$\text{Eq2} \rightarrow \text{Unary ListaT} ; \mid \text{ListaT} ;$

$\text{ListaT} \rightarrow \text{T ListaT2}$

$\text{ListaT2} \rightarrow \text{T ListaT2} \mid \epsilon$

$\text{T} \rightarrow \text{Operando T2}$

$\text{T2} \rightarrow \text{Binary Operando T2} \mid \epsilon$

$\text{Binary} \rightarrow +$

$\text{Binary} \rightarrow -$

$\text{Binary} \rightarrow *$

$\text{Binary} \rightarrow /$

$\text{Operando} \rightarrow \text{id}$

$\text{Operando} \rightarrow \text{num}$

$\text{Operando} \rightarrow ( \text{T} )$

$\text{Unary} \rightarrow +$

$\text{Unary} \rightarrow -$