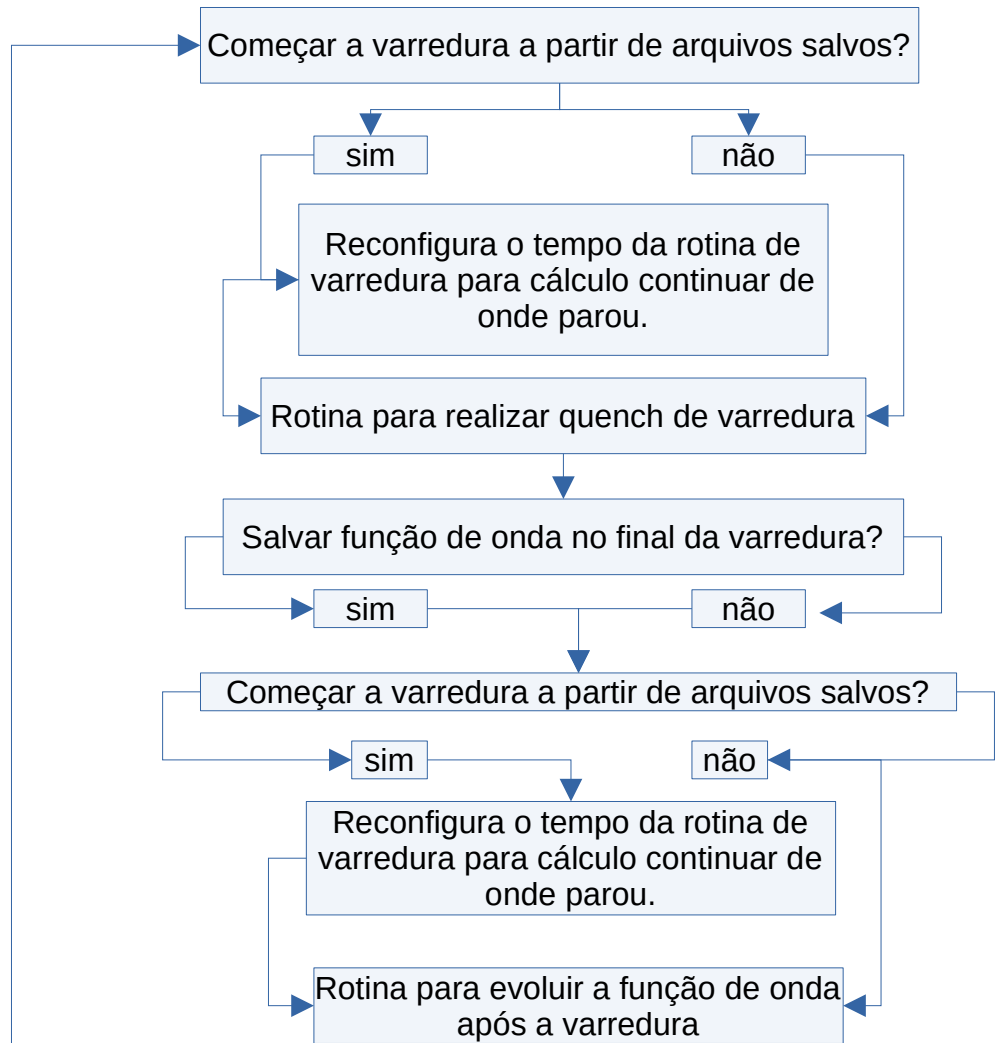
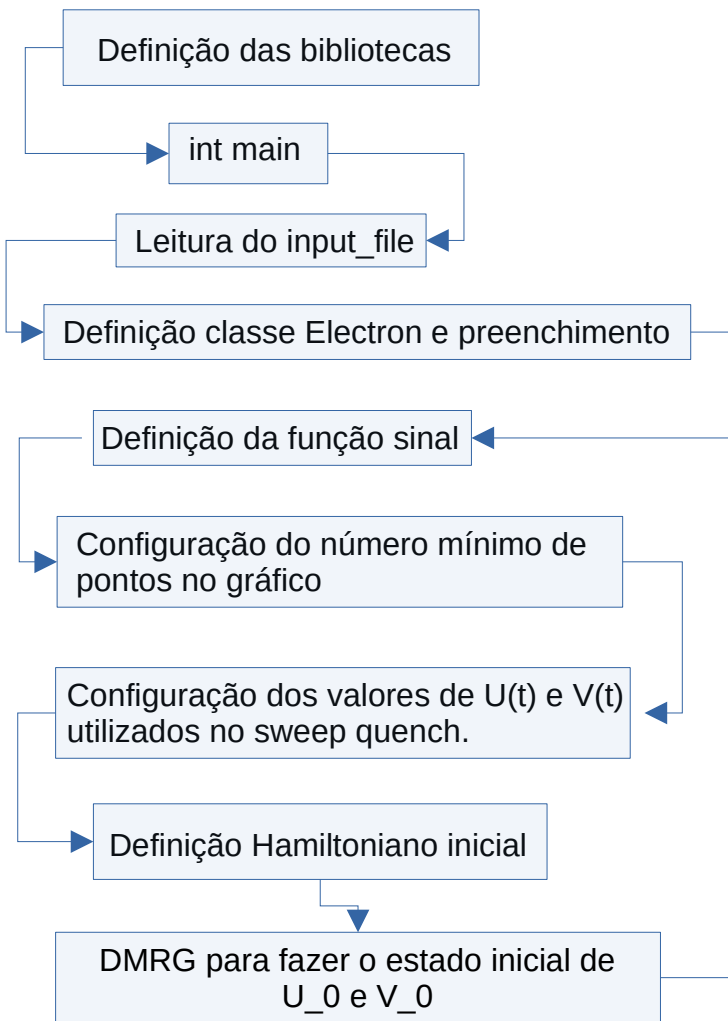


Design Document

Isaac Martins Carvalho

Macrodivisão do código

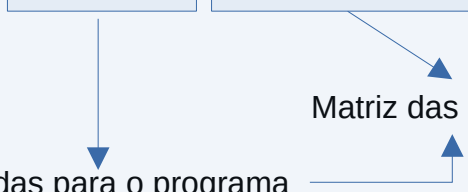


Função **main** e leitura do input

```
int main(int argc, char* argv[])  
{
```

Entradas para o programa

Matriz das entradas



- **argv[0]**: nome do programa.
- **argv[1]**: primeira entrada não trivial na linha de comando.

Ao rodar o código digitando no terminal

```
> ./programa input_file
```

teremos

- **argv[0]** == "programa"
- **argv[1]** == "input_file"

```
if(argc != 2)  
{  
    printfln("Usage: %s inputfile", argv[0]);  
    return 0;  
}
```

A declaração **if(argc != 2)** garante que fornecemos as duas entradas: **argv[0]** e **argv[1]**.

```
auto input = InputGroup(argv[1], "input");
```

InputGroup: importa a entrada de **argv[1]** em **input**

Importando dados do **Input_file** no código

Para entradas do tipo **int**, exemplo:

```
auto Npart = input.getInt("Npart", L);
```

Lê "Npart" do **input** e armazena em **Npart**. Caso "Npart" não esteja definido em **input**, retorna **L** (opcional).

Para entradas do tipo **Real**, exemplo:

```
auto t1 = input.getReal("t1", 1.);
```

Lê "t1" do **input** e armazena em **t1**. Caso "t1" não esteja definido em **input**, retorna **1** (opcional).

Para entradas do tipo **Yes ou No**, exemplo:

```
auto quiet = input.getYesNo("quiet", false);
```

Lê "quiet" do **input** e armazena em **quiet**. Caso "quiet" não esteja definido em **input**, retorna **false** (opcional).

Definição classe Electron e preenchimento

```
auto sites = Electron(L);
```

Classe para especificar o espaço de Hilbert. Armazena apenas o número de sites e informações sobre cada site. A classe Electron representa férmions de spin $\frac{1}{2}$ ocupando um único orbital. A ocupação máxima por site é duas partículas.

```
auto state = InitState(sites);
```

Classe para inicializar o MPS. Para sites do tipo Electron comporta a seguintes string de estado:

States of a ElectronSite

- "Emp" — the vacuum (empty) state (alternate name "0")
- "Up" — site occupied by one spin up particle (alternate name "+")
- "Dn" — site occupied by one spin down particle (alternate name "-")
- "UpDn" — site occupied by two particles, one of each spin (alternate name "S" for singlet)

A entrada **state** será utilizada para configurar o preenchimento da cadeia.

Definição classe Electron e preenchimento

Configurando o preenchimento

```
int p = Npart; //Filling
//! Funcao para configurar o preenchimento da cadeia
for(int i = L; i >= 1; --i)
{
    if(p > i)
    {
        println("Doubly occupying site ",i);
        state.set(i,"UpDn");
        p -= 2;
    }
    else
    if(p > 0)
    {
        println("Singly occupying site ",i);
        state.set(i,(i%2==1 ? "Up" : "Dn"));
        p -= 1;
    }
    else
    {
        state.set(i,"Emp");
    }
}
```

state.set(i,"UpDn");

Configura a ocupação do sítio i para dupla ocupação **UpDn**

Descrição da rotina de preenchimento

- 1) Configura-se a entrada p para receber o preenchimento;
- 2) Caso $p > i$, configura-se o preenchimento do site i para **UpDn** e subtrai $p - 2$.
- 3) Se 2) falso e $p > 0$, configura-se o preenchimento do site i para **Up** se i ímpar e **Dn** se i par, depois subtrai $p-1$.
- 4) Se 2) e 3) falso, então o sítio i recebe ocupação vazia **Emp**.

Contrói o MPS usando states e salva em psi0

```
auto psi0 = MPS(state); ///< Salva o estado inicial em psi0, utilizado para o DMRG a seguir.
```

Definição da função sinal

Parte da definição de $U(t)$ e $V(t)$ inclui a função sinal. No código ela está definida como as entradas `sgn_U` e `sgn_V`:

```
#!/ Definção da função sgn
auto sgn_U = Uf-U0;
auto sgn_V = Vf-V0;

if(sgn_U < 0.){sgn_U = -1.;}
if(sgn_U == 0.){sgn_U = 0.;}
if(sgn_U > 0){sgn_U = 1.;}

if(sgn_V < 0.){sgn_V = -1.;}
if(sgn_V == 0.){sgn_V = 0.;}
if(sgn_V > 0){sgn_V = 1.;}

```

RELEMBRANDO

Iremos considerar o hamiltoniano com função dos parâmetros internos U e V , tal que $\hat{H}(U, V)$. Usaremos a notação (U_0, V_0) para identificar os parâmetros iniciais e (U_F, V_F) para identificar os parâmetros finais. Para que o quench de (U_0, V_0) para (U_F, V_F) seja adiabático, devemos atualizar os valores de $U(t)$ e $V(t)$ tal que a variação da energia interno do sistema à medida que o sistema evolui seja suficientemente pequena. Desta forma iremos usar a definição

$$U(t) = U_0 + \operatorname{sgn}(U_F - U_0) \frac{t}{\tau_U} \quad \text{e} \quad (2.1)$$

$$V(t) = V_0 + \operatorname{sgn}(V_F - V_0) \frac{t}{\tau_V}, \quad (2.2)$$

onde τ_U e τ_V controlam a velocidade na qual $U(t)$ e $V(t)$ variam no tempo, respectivamente, e $0 < t < |U_F - U_i| \tau_U$ $0 < t < |V_F - V_i| \tau_V$. Como em nosso quench consideramos a variação de dois parâmetros, U e V , é necessário configurarmos τ_U e τ_V para que o tempo final seja o mesmo, ou seja, $|U_F - U_i| \tau_U = |V_F - V_i| \tau_V$. Isso nos leva a relação

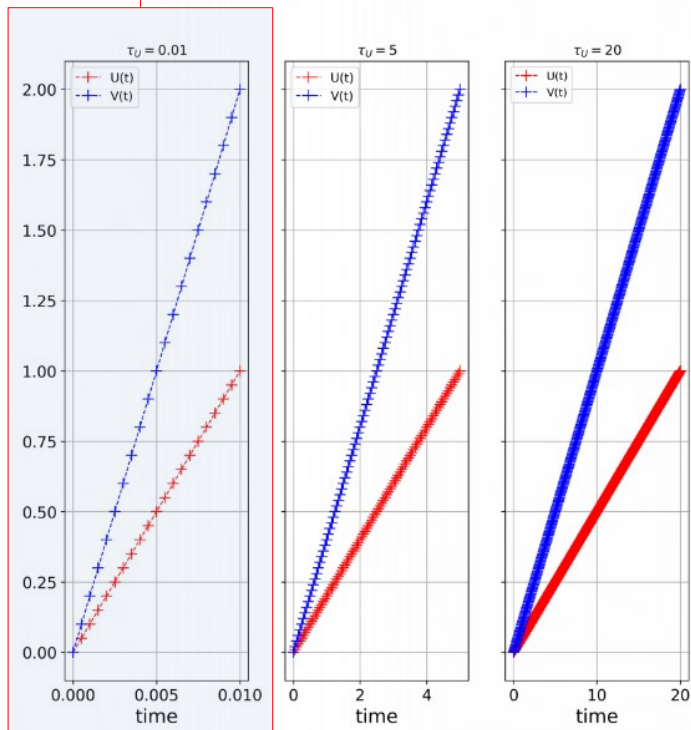
$$\tau_V = \frac{|U_F - U_i|}{|V_F - V_i|} \tau_U. \quad (2.3)$$

Configuração do número mínimo de pontos no gráfico

No quench de varredura, fracionamos o tempo de varredura $0 < t < |U_f - U_0| \tau_U$ em termos da precisão $tstep$, definida no input. Esse fracionamento é feito porque ao longo da varredura os valores de $U_0 \leq U(t) \leq U_f$ e $V_0 \leq V(t) \leq V_f$ são atualizados no tempo t .

Exemplo do fracionamento de $U(t)$ e $V(t)$ em termos de $tstep$:

$U_0 = 0.0$ and $V_0 = 0.0$ to $U_f = 1.0$ and $V_f = 2.0$



Como é fracionado?

$$\frac{|U_f - U_0| \tau_U}{tstep} = \text{num de pontos gráfico}$$

Para o caso de $\tau_U = 0.01$, se configuramos $tstep = 0.01$, teríamos apenas um ponto no gráfico do painel de $\tau_U = 0.01$.

Para os demais painéis de $\tau_U = 5$ e 20 , configurar $tstep = 0.01$ oferece mais de 20 pontos no gráfico

O número de pontos no gráfico define o número de evolução que iremos realizar para completar a varredura. A cada instante de tempo t da evolução atualizamos $U(t)$ para $U(t + tstep)$ e $V(t)$ para $V(t + tstep)$ até os parâmetros finais U_f e V_f .

O quench de varredura é chamado de linear porque $U(t)$ e $V(t)$ variam linearmente no tempo t , como visto na figura ao lado.

Configuração do número mínimo de pontos no gráfico

Como está no código?

```
double time_f = (sqrt(pow(Uf - U0,2))*tau_U); ///< time_f e o tempo total do sweep quench
println("\ntime_f= ", time_f);

//! Numero de pontos minimos para o sweep quench.
//!
//! Condicao para termos ao menos 20 pontos no grafico (p_t = 20 abaixo).
//! Diminui a precisao tstep se tau_U for muito pequeno
//!
int p_t = 20; ///

- 1) A variável time_f recebe o tempo total da varredura.
- 2) A variável p_t recebe o número mínimo de pontos no gráfico.
- 3) Se  $\text{time\_f}/\text{tstep} < p_t$  então  $\text{tstep} = \text{time\_f}/p_t$ .
- 4) A variável n_loop, que determina o número de evoluções da rotina de evolução, recebe o valor inteiro de  $\text{time\_f}/\text{tstep}$ .

```

Para o caso onde $\text{time_f} \neq n_loop * \text{tstep}$

Esse cenário pode acontecer quando o resto da divisão de time_f por tstep não é igual a zero. Corrigimos isso, reconfigurando o tstep para o valor mais próximo definido pelo usuário.

```
if(time_f != n_loop*tstep)
{
    tstep = time_f/n_loop;
    println("tstep foi reconfigurado para = ",tstep);
}
```


Configuração do valores de U(t) e V(t) utilizados no sweep quench

Como funciona?

```
/*! Definição de U(t) e V(t) utilizados no sweep quench.
*/
O valor de U(t) e V(t) são atualizados em cada step de tempo da varredura.
A função abaixo configura um valor para U(t) e V(t) em cada step de
tempo.
*/

Real del_t = tstep; ///< Valor de del_t utilizado para construir os vetores U(t) e V(t).

Vector U_t(n_loop + 1) , V_t(n_loop + 1);

for(int j = 0; j <= n_loop; j += 1)
{
    double d_t = j*del_t;

    U_t(j) = U0 + (sgn_U*d_t)/tau_U;
    //cout << "U_t(" << j << ") = " << U_t(j) << "\n";
}
for(int j = 0; j <= n_loop; j += 1)
{
    double d_t = j*del_t;

    V_t(j) = V0 + (sgn_V*d_t)/tau_V;
    //cout << "V_t(" << j << ") = " << V_t(j) << "\n";
}
```

Definimos a variável `del_t` para receber `tstep`, determinado anteriormente. Criamos dois vetores, `U_t()` e `V_t()`, os quais possuem o tamanho `n_loop + 1`. O tamanho `n_loop + 1`, ao invés de `n_loop` somente, foi escolhido porque iremos configurar `U_t(0) = U_0` e `V_t(0) = V_0`.

Utilizamos a função sinal `sgn` definida anteriormente, para escrever os valores de `U(t)` e `V(t)`, descritos pelas equações:

$$U(t) = U_0 + \text{sgn}(U_F - U_0) \frac{t}{\tau_U} \text{ e}$$
$$V(t) = V_0 + \text{sgn}(V_F - V_0) \frac{t}{\tau_V},$$

A rotina funciona atribuindo os valores de `U_t` e `V_t` para cada instante `t` da varredura.

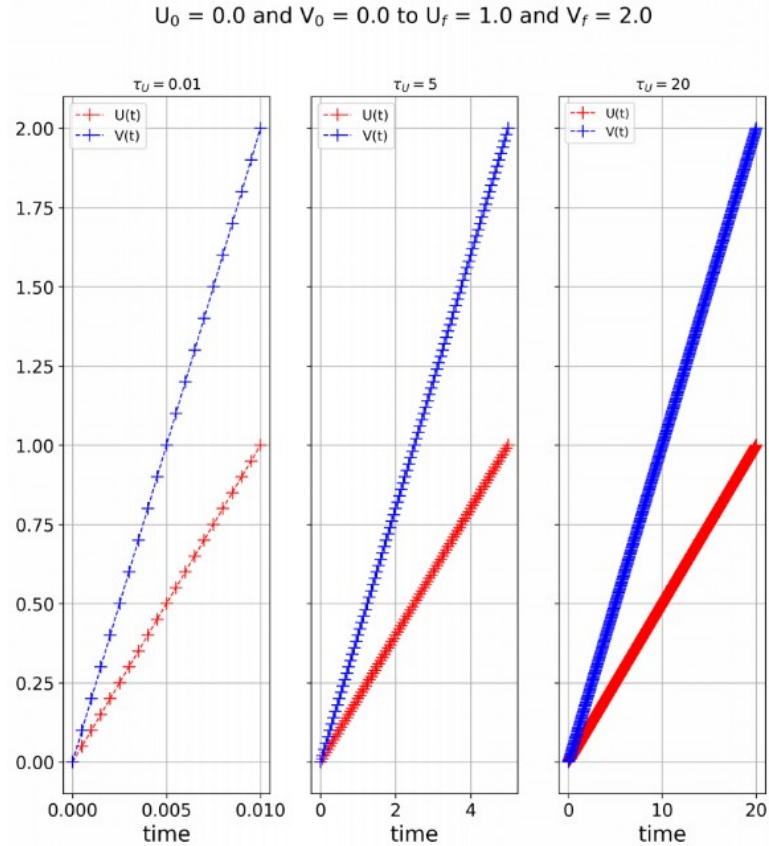
IMPORTANTE!!!

```
auto tau_V = (tau_U*(sqrt(pow(Uf-U0,2)))/(sqrt(pow(Vf-V0,2)))); ///< Definição de tau_U
```

O valor de `tau_U` é definido no `input_file`, mas o valor de `tau_V` é avaliado na seção “Leitura do input” do código:

$$\tau_V = \frac{|U_F - U_0|}{|V_F - V_0|} \tau_U$$

Configuração do valores de $U(t)$ e $V(t)$ utilizados no sweep quench



Os valores de U_t e V_t são determinados pelos pares (U_0, V_0) , (U_f, V_f) e por τ_U . A figura ao lado trás um exemplo dos valores instantâneos assumidos por essas quantidades ao longo do time de varredura.

Definição Hamiltoniano inicial

```
#!/Ground State (GS) Hamiltonian
/*!
    Definicao do Hamiltoniano do estado inicial.
    Esse Hamiltoniano sera utilizado para realizar DRMG a fim
    de rastrear o estado fundamental com a maior precisao possivel.
*/
auto ampo = AutoMPO(sites); ///
```

MPO é uma classe para armazenar operadores de produtos de matrizes, que são semelhantes aos MPS.

Escrever MPO pode ser algo muito técnico. O **AutoMPO** é um ferramenta que possui a vantagem de produzir o MPO automaticamente de forma simples a partir de entradas legíveis. Depois de obter o MPO, ele pode ser usado como entrada para um cálculo DMRG ou avaliar observáveis

operador **AutoMPO +=** para um sítio aceita o input

[value], "[operator name]", [site]

operador **AutoMPO +=** para dois sítio aceita o input

[value], "[operator name]", [site], "[operator name]", [site]

Definição Hamiltoniano inicial

```
#!/Ground State (GS) Hamiltonian
/*!
  Definicao do Hamiltoniano do estado inicial.
  Esse Hamiltoniano sera utilizado para realizar DRMG a fim
  de rastrear o estado fundamental com a maior precisao possivel.
*/
auto ampo = AutoMPO(sites); ///
```

Modelo implementado:

$$H = -t_h \sum_{j=1}^L \sum_{\sigma=\uparrow,\downarrow} \left(a_{j,\sigma}^\dagger a_{j+1,\sigma} + H.c. \right) + U \sum_{j=1}^L \hat{n}_{j\uparrow} \hat{n}_{j\downarrow} + V \sum_{j=1}^L \hat{n}_j \hat{n}_{j+1},$$

Operators Provided by ElectronSite

- "Nup" — density of up-spin particles \hat{n}_\uparrow
- "Ndn" — density of down-spin particles \hat{n}_\downarrow
- "Nupdn" — density of doubly-occupied sites $\hat{n}_\uparrow \hat{n}_\downarrow$
- "Ntot" — the total density operator $\hat{n}_{\text{tot}} = \sum_{\sigma} \hat{n}_{\sigma}$
- "Aup" — the up-spin annihilation operator \hat{a}_\uparrow
- "Adagup" — the up-spin creation operator \hat{a}_\uparrow^\dagger
- "Adn" — the down-spin annihilation operator \hat{a}_\downarrow
- "Adagdn" — the down-spin creation operator $\hat{a}_\downarrow^\dagger$
- "F" — the Jordan-Wigner fermion 'string' operator $\hat{F} = (-1)^{\hat{n}_{\text{tot}}}$
- "Sz" — the z-component spin operator (matrix elements +0.5 for an up spin, -0.5 for a down spin)
- "S+" — the spin raising operator (matrix element +1.0 for mapping a down spin to

Para entendimento e manipulação dos operadores da classe ElectronSite recomendo fortemente a leitura dos dois links a seguir

Para entender quando usar as string de férmions:

<https://itensor.org/docs.cgi?page=tutorials/fermions>

Para entender a classe ElectronSite:

<https://itensor.org/docs.cgi?vers=cppv3&page=classes/electron>

O recurso AutoMPO já ajusta as strings de férmions automaticamente, mas para MPO que não utilizam esse recurso, as strings devem ser colocadas. Por exemplo, quando fizemos os gates da evolução t-DRMG iremos usá-las.

Começar a varredura a partir de arquivos salvos?

Essa rotina está inclusa no quench de varredura:

```
////////////////////  
/// Finite time quantum quench  
////////////////////  
  
/*!  
  Rotina para os sweep quench. Nessa rotina esta inclusa  
  uma serie de condicoes, localizada no escopo de cada medida,  
  que possuem a finalidade de controlar o que sera medido durante a varredura.  
  A configuracao do que sera medido se encontra no arquivo input_file.  
  Alem disso, e avaliado o tempo das rotinas DRMG e t-DRMG em cada instante da varredura.  
*/
```

```
////////////////////  
/// Rotina para salvar psi e sites na varredura  
////////////////////  
  
/*!  
  No input_file configura-se o intervalo de tempo em que os arquivos de psi serao salvos por meio  
  da entrada per_save. A entrada per_save corresponde ao valor percentual no qual deseja-se salvar os arquivos.  
  Exemplo: se per_save = 10, então a cada 10% do tempo total do sweep quench os arquivos serao salvos.  
*/
```

É necessário salvar os arquivos de **sites** e dos **psi's** para reiniciar o cálculo posteriormente.

Por que não salvar só os arquivos de **psi**?

Os arquivos de **site** e **psi** devem ser armazenados juntos porque durante o cálculo eles são montados utilizando os mesmos índices dos números quânticos relacionados a representação do estado. Quando reiniciamos o cálculo com **psi** sem os **sites** que o projetou, os índices dos **sites** utilizados no cálculo em vigor não reconhecem os índices de **psi**. Quando isso ocorre o output do terminal apresenta um erro relacionado a representação dos números quânticos utilizados na representação de **psi**.

Começar a varredura a partir de arquivos salvos?

Descrição da rotina para salvar os arquivos de **sites** e **psi**:

- Na seção “Configuração do número mínimo de pontos no gráfico” vimos que número de pontos do gráfico, **n_loop**, determina o número de evoluções **tstep** que iremos realizar para completar a varredura de **U_0** e **V_0** para **U_f** e **V_f**. Por exemplo, o tempo total da varredura pode ser indentificado por **tstep * n_loop**, e tempos intermediário por **t' * tstep**, sendo **t'** um número inteiro pertencente ao intervalo **0 <= t' < n_loop**.
- Na rotina de salvar os arquivos da varredura começa com o usuário escolhendo a taxa percentual do tempo total no qual os arquivos serão salvos. Por exemplo, se o usuário escolher 10%, então a cada 10% de **n_loop** os arquivos serão salvos em pasta. No input configuramos a variável **per_save** para definir esse percentual.
- No input o usuário determina **per_save** como um número inteiro de 0 a 100.
- Salvamos o percentual correspondente de **n_loop** em **psave**:

```
int psave = round((per_save*n_loop)/100); //round arredonda o valor
printfln("\npsave = ",psave);
```

Começar a varredura a partir de arquivos salvos?

O vetor criado nessa rotina será utilizado no loop de varredura para salvar os arquivos nos tempos determinados.

A rotina funciona criando um vetor `arq_save` de dimensão **`n_loop + 1`**. Configuramos a entradas de **`arq_save`** para receber a taxa **`psave`** no qual os arquivos serão salvos.

```
//criando um vetor arq_save que sera utilizado para ler e salvar os arquivos
Vector arq_save(n_loop+1);

int e1 = psave;

for(int j = 0; j <= n_loop; j += 1)
{
    if(j == e1)
    {
        arq_save(j) = e1;
        e1 += psave;
    }
    //para comparar arq_save com o tempo de varredura
    cout << "arq_save(" << j << ") = " << arq_save(j) << "    " << "sweep_time = " << j*del_t << "\n";
}
```

Utilizaremos o vetor **`arq_save`** para construir a condição de salvar arquivos durante o loop de varredura.

Exemplo do output para `per_save=10` no quench de `U_0=V_0=0` para `U_f=1` e `V_f=2`, sendo `tau_U=0.01`.

```
arq_save(0) = 0    sweep_time = 0
arq_save(1) = 0    sweep_time = 0.0005
arq_save(2) = 2    sweep_time = 0.001
arq_save(3) = 0    sweep_time = 0.0015
arq_save(4) = 4    sweep_time = 0.002
arq_save(5) = 0    sweep_time = 0.0025
arq_save(6) = 6    sweep_time = 0.003
arq_save(7) = 0    sweep_time = 0.0035
arq_save(8) = 8    sweep_time = 0.004
arq_save(9) = 0    sweep_time = 0.0045
arq_save(10) = 10   sweep_time = 0.005
arq_save(11) = 0    sweep_time = 0.0055
arq_save(12) = 12   sweep_time = 0.006
arq_save(13) = 0    sweep_time = 0.0065
arq_save(14) = 14   sweep_time = 0.007
arq_save(15) = 0    sweep_time = 0.0075
arq_save(16) = 16   sweep_time = 0.008
arq_save(17) = 0    sweep_time = 0.0085
arq_save(18) = 18   sweep_time = 0.009
arq_save(19) = 0    sweep_time = 0.0095
arq_save(20) = 20   sweep_time = 0.01
```

Começar a varredura a partir de arquivos salvos?

Rotina para começar a varredura a partir de arquivos salvos

```
int l_condi = 0; //utilizado para condicionar o inicio do loop de varredura, depende do valor de tvarre

//Se o usuario setar um valor diferente de zero no input, a funcao le os arquivos da pasta
if(tvarre != 0.)
{

    //definindo int para loop condicionado a onde o calculo parou
    l_condi = tvarre/del_t;

    printfln("l_condi = ", l_condi);

    string s_tl = format("sites_varredura_L_", L, "_Npart_", Npart, "_V0_", V0, "_U0_", U0, "_Uf_", Uf, "_Vf_", Vf, "_tau_U_", tau_U, "_tvarre_", tvarre);
    string p_tl = format("psi_varredura_L_", L, "_Npart_", Npart, "_V0_", V0, "_U0_", U0, "_Uf_", Uf, "_Vf_", Vf, "_tau_U_", tau_U, "_tvarre_", tvarre);
    readFromFile(s_tl, sites); ///< Le na pasta local os sites
    psi_Evol = readFromFile<MPS>(p_tl); ///< Le na pasta local a função de onda do tempo tevolu.
} //if(tvarre != 0.)
```

Se o usuário configurar um valor **tvarre**, definida como o tempo para iniciar a varredura, diferente de 0.0, então os arquivos referente ao tempo **tvarre** são lidos e a variável **l_condi** que condiona o número de interações no loop de varredura é reconfigurada. Caso o usuário configure **tvarre=0.0**, o loop de varredura inicia em **l_condi=0**, o que é equivalente a iniciar a varredura no tempo igual a zero.

```
////////////////////////////////////
// Inicio do loop de varredura
////////////////////////////////////

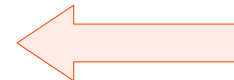
for(int lt = l_condi; lt <= n_loop; lt ++){
    .....
}
```


Rotina para realizar quench de varredura

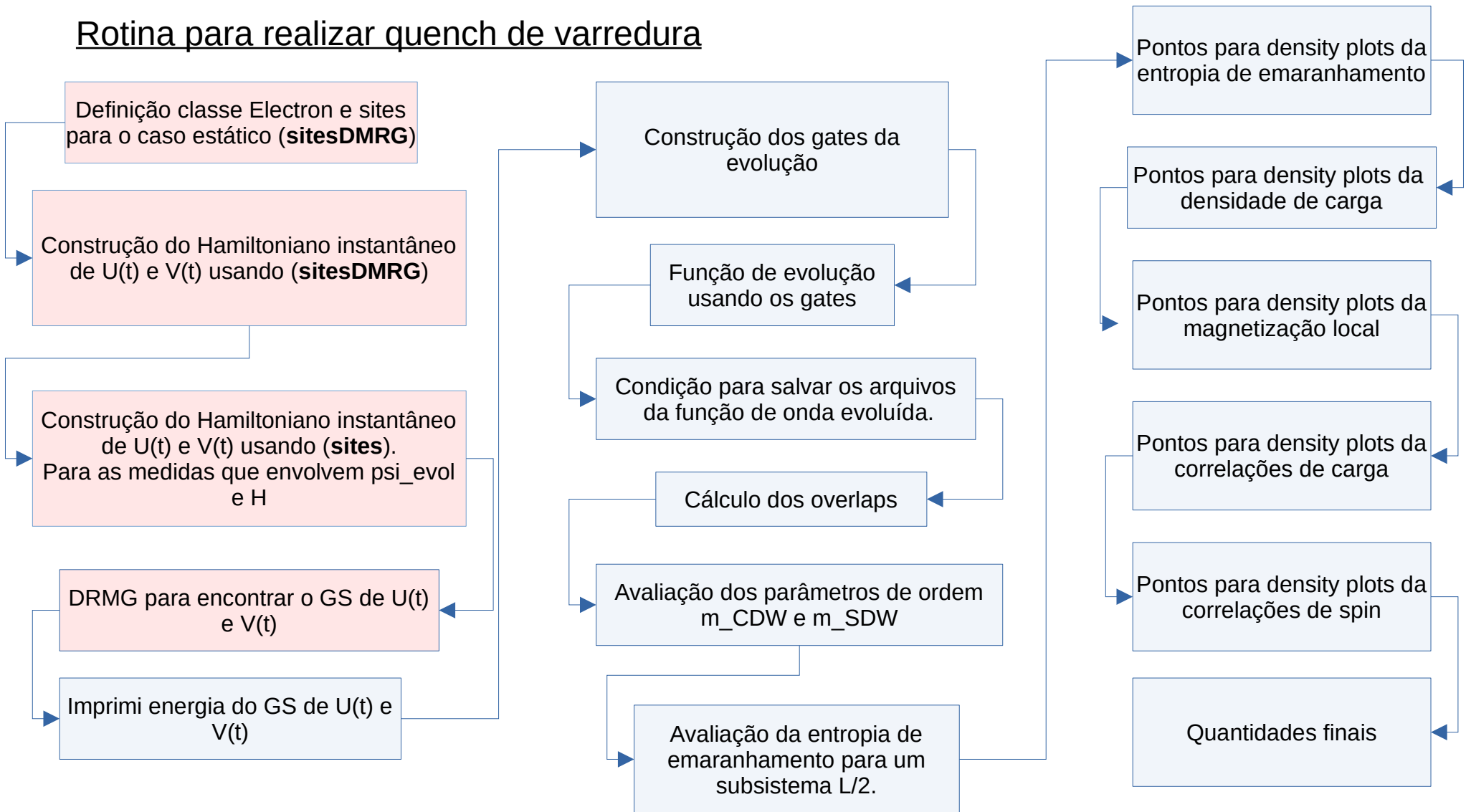
Considerações iniciais

- Nessa rotina o usuário pode escolher se deseja avaliar o GS instantâneo de $U(t)$ e $V(t)$ a fim de comparar com os resultados para o estado de não-equilíbrio.
- As configurações do que será medido durante a varredura são feitas no input.
- A classe `sites`, utilizada para avaliar as quantidades de equilíbrio, foram reconstruída sobre outra entrada, **`sitesDMRG`**, a fim de evitar conflitos com os arquivos de `site` salvos para os estados de não-equilíbrio, lidos em cálculo reiniciados.
 - Para realizar DMRG e avaliar as quantidades de equilíbrio durante o loop de varredura utilizaremos **`sitesDRMG`**.
- Como há quantidades que consideram o sanduíche da função de onda evoluída com o Hamiltoniano, foi necessário construir um Hamiltoniano usando a entrada **`site`**, a mesma utilizada pela função de onda evoluída. Do contrário o cálculo apresenta um erro ao reiniciar a partir de arquivos salvos.

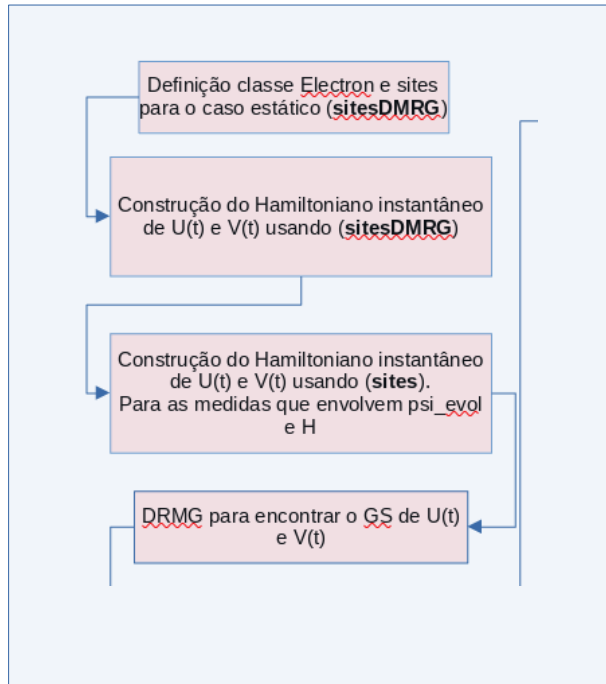
A seguir apresentamos o fluxograma da rotina do loop de varredura. As caixas do fluxograma em cor vermelha são para as rotinas já descritas anteriormente neste documento, enquanto as caixas em azul serão descritas na sequência.



Rotina para realizar quench de varredura



Rotina para realizar quench de varredura



A descrição das caixas em vermelho podem ser vistas nos slides anteriores.

No início do documento havíamos definido entradas para as classes **InitialState** e **Electron** a fim de construir a função de onda inicial utilizada como input do cálculo DMRG. No loop de varredura foi necessário redefinir essas funções sobre outras entradas, pois no caso das rotinas iniciadas a partir de arquivos salvos havia problemas de conflito entre os índices de funções definidas sobre a entrada de **sites** do cálculo atual e os índices dos **sites** reiniciados.

Imprimir energia do GS de $U(t)$ e $V(t)$

A função DMRG presente na varredura pode ser modificada para imprimir os detalhes do cálculo em cada sweep. Para fazer isso é necessário modificar {"Quiet", true} → {"Silent", true}. No entanto esses detalhes podem tornar a análise difícil, ainda mais pelo fato de existir vários loops na rotina de varredura. Uma forma rápida de sabermos a energia ao final do quench printar **energy_GS**, presente no output do cálculo DMRG.



```
#!/Realiza DMRG, retorna energy_GS e psi_GS
auto [energy_GS,psi_GS] = dmrg(H,psi0,sweeps,{"Quiet",true}); ///
```

```
println("Ground state energy");
cout << "%energy_GS " << sqrt(pow(U_t(lt)-U0,2)) << " " << energy_GS << "\n";
cout << "%energy_GS_t " << lt*del_t << " " << energy_GS << "\n";
```

Construção dos gates da evolução

Para o entendimento da evolução de um MPS (em nosso caso a função de onda) utilizando Trotter Gates recomendamos a leitura dos links:

https://itensor.org/docs.cgi?vers=cppv3&page=formulas/tevol_trotter

https://itensor.org/docs.cgi?vers=cppv3&page=formulas/gates_to_mpo

<https://iopscience.iop.org/article/10.1088/1367-2630/12/5/055026/pdf> (background teórico)

Esse trecho extraído do site da Itensor resume a ideia básica.

If the Hamiltonian is a sum of local terms

$$H = \sum_j h_{j,j+1}$$

where $h_{j,j+1}$ only acts non-trivially on sites j and $(j+1)$, then a Trotter decomposition that is particularly well suited for use with MPS techniques is

$$e^{-i\tau H} \approx e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2} + O(\tau^3)$$

Note the factors of two in each exponential. The error in the above decomposition is of order τ^3 , so this will be the error accumulated *per time step*. Because of the time-step error, one takes τ to be small and then applies the above set of operators to an MPS as a single sweep, then does a number (t/τ) of sweeps to evolve for a total time t . The total error will therefore scale as τ^2 with this scheme, though other sources of error may dominate for long times, or very small τ , such as truncation errors.

The same decomposition can be used for imaginary time evolution just by replacing $i\tau \rightarrow \tau$.

Construção dos gates da evolução

Definição das entradas necessárias para construção dos gates

- *tstep* : step de tempo (quanto menor mais preciso o cálculo);
- *ttotal* : tempo total da evolução e
- *cutoff* : erro de truncagem da evolução.

Início da função no código que faz a evolução:

```
//!Construção dos gates da evolução
/*!
    Abaixo é construído os gates utilizados na decomposição Trotter.
    Esses gates irão compor a função que realiza as evoluções do sweep quench.
*/
//Building gates of evolution

// Cria um std :: vector (array dinamicamente dimensionável)
// para armazenar os Trotter gates
auto gates = vector<BondGate>();
```

Quando configuramos o Hamiltoniano não foi necessário incluir as string para férmions, pois a classe **AutoMPO** incluir essa utilidade quando declaramos que os sites são do tipo **Electron**. Por outro lado, para os gates devemos incluir manualmente essas strings a fim de manter a simetria fermiônica durante a evolução, produzindo os resultados corretos. Duas classes serão utilizadas nessa construção:

BondGate: aceita os termos do Hamiltoniano local $h_{j,j+1}$ como um tensor. Esse tensor é automaticamente exponenciado para construir o gate de Trotter com um *tstep* específico.

gateTEvol: função que aplica automaticamente os gates construídos no MPS (em nosso caso representado pela função de onda), realizando a evolução da função.

Construção dos gates da evolução

$$e^{-i\tau H} \approx e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2} + O(\tau^3)$$

A ideia é construir de expansão de $\exp(-i\tau H)$ incluindo os operadores $\exp(-i\tau h_{\text{term}}/2)$ aos gates. Nesse primeiro trecho do código nos iremos construir os gates da parte:

$$e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2}$$

que vai do sítio 1 até o sítio L-1. Iremos na sequência adicionar esses gates a variável **gates** definida anteriormente.

Os operadores locais (operadores relacionados ao sítio) são incluídos ao **hterm** por meio da função **sites.op("operador",site)** como no exemplo abaixo:

```
auto hterm = -t1*sites.op("Adagup*F",b)*sites.op("Aup",b+1);
```

Basicamente o que estamos fazendo é escrever o Hamiltoniano locais de:

$$H = -t_h \sum_{j=1}^L \sum_{\sigma=\uparrow,\downarrow} (a_{j,\sigma}^\dagger a_{j+1,\sigma} + H.c.) + U \sum_{j=1}^L \hat{n}_{j\uparrow} \hat{n}_{j\downarrow} + V \sum_{j=1}^L \hat{n}_j \hat{n}_{j+1},$$

mas utilizando as string de férmions "F", como descrito em <https://itensor.org/docs.cgi?page=tutorials/fermions>, pois os operadores de campo **A** da ITensor são definidos para bóson. O uso da string é necessário para obtermos um resultado correto.

```
//Cria os gates exp(-i*step/2*hterm)
//e os adiciona ao "gates"
|
for(int b = 1; b < L; ++b)
{
    auto hterm = -t1*sites.op("Adagup*F",b)*sites.op("Aup",b+1);
    hterm += -t1*sites.op("Adagdn",b)*sites.op("F*Adn",b+1);
    hterm += t1*sites.op("Aup*F",b)*sites.op("Adagup",b+1);
    hterm += t1*sites.op("Adn",b)*sites.op("F*Adagdn",b+1);
    hterm += V_t(lt)*sites.op("Ntot",b)*sites.op("Ntot",b+1);
    hterm += U_t(lt)*sites.op("Nupdn",b)*sites.op("Id",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
for(int b = L-1; b <= L-1; ++b)
{
    auto hterm = U_t(lt)*sites.op("Id",b)*sites.op("Nupdn",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
for(int b = L-1; b <= L-1; ++b)
{
    auto hterm = h_z*sites.op("Id",b)*sites.op("Sz",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```

Construção dos gates da evolução

$$e^{-i\tau H} \approx e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2} + O(\tau^3)$$

Para escrever os termos dos Hamiltonianos locais:

Next-neighbor "hopping" part of a 1d fermionic Hamiltonian:

$$\sum_{\sigma} (c_{\sigma,i}^{\dagger} c_{\sigma,i+1} + c_{\sigma,i+1}^{\dagger} c_{\sigma,i}) = (a_{\uparrow,i}^{\dagger} F_i) a_{\uparrow,i+1} + a_{\downarrow,i}^{\dagger} (F_{i+1} a_{\downarrow,i+1}) - (a_{\uparrow,i} F_i) a_{\uparrow,i+1}^{\dagger} - a_{\downarrow,i} (F_{i+1} a_{\downarrow,i+1}^{\dagger})$$

$$V \hat{n}_j \hat{n}_{j+1}$$

$$U \hat{n}_{j\uparrow} \hat{n}_{j\downarrow}$$

Os gates são operadores de dois sítios, por esse motivo incluímos o operador identidade "Id" quando o operador é definido somente sobre um sítio, como no caso das interações onsite.

O segundo **for** (loop do recorte) realiza somente uma repetição para b=L-1 para incluir as interações onsite no sítio b+1.

Explicado no próximo slide

```
//Cria os gates exp(-i*tstep/2*hterm)
//e os adiciona ao "gates"
```

```
|
for(int b = 1; b < L; ++b)
```

```
{
    auto hterm = -t1*sites.op("Adagup"F",b)*sites.op("Aup",b+1);
    hterm += -t1*sites.op("Adagdn",b)*sites.op("F*Adn",b+1);
    hterm += t1*sites.op("Aup"F",b)*sites.op("Adagup",b+1);
    hterm += t1*sites.op("Adn",b)*sites.op("F*Adagdn",b+1);

    hterm += V t(lt)*sites.op("Ntot",b)*sites.op("Ntot",b+1);
    hterm += U t(lt)*sites.op("Nupdn",b)*sites.op("Id",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```

```
for(int b = L-1; b <= L-1; ++b)
```

```
{
    auto hterm = U t(lt)*sites.op("Id",b)*sites.op("Nupdn",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```

```
for(int b = L-1; b <= L-1; ++b)
```

```
{
    auto hterm = h_z*sites.op("Id",b)*sites.op("Sz",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```


Construção dos gates da evolução

$$e^{-i\tau H} \approx e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2} + O(\tau^3)$$

Trecho referente a evolução mantendo um campo fixo aplicado às extremidades da cadeia.

Novamente, os gates são operadores definidos sobre dois sítios, por esse motivo é necessário utilizar o operador “Id” para os sítios em que o campo não é aplicado.

```
for(int b = L-1; b <= L-1; ++b)
{
    auto hterm = h_z*sites.op("Id",b)*sites.op("Sz", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
for(int b = 1; b <= 1; ++b)
{
    auto hterm = h_z*sites.op("Sz",b)*sites.op("Id", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```

Construção dos gates da evolução

$$e^{-i\tau H} \approx e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2} + O(\tau^3)$$

Trecho referente a evolução mantendo um campo fixo aplicado às extremidades da cadeia.

Novamente, os gates são operadores definidos sobre dois sítios, por esse motivo é necessário utilizar o operador “Id” para os sítios em que o campo não é aplicado.

```
for(int b = L-1; b <= L-1; ++b)
{
    auto hterm = h_z*sites.op("Id",b)*sites.op("Sz", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
for(int b = 1; b <= 1; ++b)
{
    auto hterm = h_z*sites.op("Sz",b)*sites.op("Id", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```

Construção dos gates da evolução

$$e^{-i\tau H} \approx e^{-ih_{1,2}\tau/2} e^{-ih_{2,3}\tau/2} \dots e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2} + O(\tau^3)$$

Para terminarmos a construção do gate da equação acima, precisamos terminar a implementação dos gates. O próximo trecho recortado do código, corresponde ao segundo trecho da equação acima:

$$e^{-ih_{N-1,N}\tau/2} e^{-ih_{N-2,N-1}\tau/2} \dots e^{-ih_{1,2}\tau/2}$$

A construção dos gates de $\exp(-i\tau\text{step}/2\text{hterm})$ deve ser feito na ordem inversa, adicionando os gates construídos a variável **gates**.

A descrição dos termos foi feita nos slides anteriores, por esse motivo não explicamos novamente aqui. Basicamente estamos escrevendo os Hamiltonianos locais correspondentes ao trecho da equação destacada acima e adicionando-os a variável **gates**. Repare que as duas construções são similares, mudando somente o sentido da estrutura de repetição para completar a construção da equação em verde.

```
//Cria os gates  $\exp(-i\tau\text{step}/2\text{hterm})$  na ordem
//reversa e adiciona-os aos "gates".

for(int b = L-1 ; b >= 1; --b)
{
    auto hterm = -t1*sites.op("Adagup"F,b)*sites.op("Aup",b+1);
    hterm += -t1*sites.op("Adagdn",b)*sites.op("F*Adn",b+1);
    hterm += t1*sites.op("Aup"F,b)*sites.op("Adagup",b+1);
    hterm += t1*sites.op("Adn",b)*sites.op("F*Adagdn",b+1);
    hterm += V_t(lt)*sites.op("Ntot",b)*sites.op("Ntot", b+1);
    hterm += U_t(lt)*sites.op("Nupdn",b)*sites.op("Id",b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
for(int b = L-1; b >= L-1; --b)
{
    auto hterm = U_t(lt)*sites.op("Id",b)*sites.op("Nupdn", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}

for(int b = L-1; b >= L-1; --b)
{
    auto hterm = h_z*sites.op("Id",b)*sites.op("Sz", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}

for(int b = 1; b >= 1; --b)
{
    auto hterm = h_z*sites.op("Sz",b)*sites.op("Id", b+1);
    auto g = BondGate(sites,b,b+1,BondGate::tReal,tstep/2.,hterm);
    gates.push_back(g);
}
```

Função de evolução usando os **gates**

```
//! Função gateTEvol que realiza a evolução de psi_Evol.  
/*!  
A função gateTEvol é definida gateTEvol(gates, tttotal, tstep, psi, {"Cutoff=", cutoff, "Verbose=", true}).  
  
https://itensor.org/docs.cgi?vers=cppv3&page=formulas/tevol\_trotter  
  
Em nosso caso, configuramos o tempo de evolução tttotal para o mesmo valor de tstep.  
O sweep quench corresponde a uma sucessão de evoluções de tempo tstep, onde em cada evolução  
os valores de  $U(t)$  e  $V(t)$  são atualizados até completar o tempo total da varredura time f.  
*/
```

```
//Função de evolução de psi_Evol, reescreve psi_Evol quando a evolução termina  
gateTEvol(gates, tstep, tstep, psi_Evol, {"Quiet", true, "Cutoff=", cutoff, "Verbose=", true, "UseSVD=", true, "SVDMethod=", "gesdd"});
```

A função **gateTEvol** aplica automaticamente os **gates** anteriormente ao MPS `psi_Evol` (correspondente ao estado evoluído da varredura). A ordem de entrada da função é:

```
gateTEvol(gates, tttotal, tstep, psi, {"Cutoff=", cutoff, "Verbose=", true});
```

mas substituímos “*tttotal*” (tempo total da evolução) para o mesmo valor de “*tstep*”, pois os valores de $U(t)$ e $V(t)$ no tempo são previamente fracionados por *tstep* (descrito na seção Configuração do número mínimo de pontos no gráfico).

Condição para salvar os arquivos da função de onda evoluída

```
//! Condição para salvar os arquivos da função de onda evoluída
if(salve_psi_durante_sweep_quench == 1)
{
    if(lt == arq_save(lt) && arq_save(lt) != 0)
    {
        string s_tl = format("sites_varredura_L",L,"Npart",Npart,"V0",V0,"U0",U0,"Uf",Uf,"Vf",Vf,"tau_U",tau_U,"tvarre",lt*del_t);
        string p_tl = format("psi_varredura_L",L,"Npart",Npart,"V0",V0,"U0",U0,"Uf",Uf,"Vf",Vf,"tau_U",tau_U,"tvarre",lt*del_t);

        writeToFile(s_tl,sites); ///< Salva na pasta local os sites
        writeToFile<MPS>(p_tl,psi_Evol); ///< Salva na pasta local a funcao de onda psi_Evol
    }
}
}

}

}

}
```

A descrição do vetor `arq_save` é feita na seção Começar a varredura a partir de arquivos salvos?. Este vetor fornece as condições para quando os arquivos de **sites** e **psi_Evol** serão salvos ao longo da varredura. A opção de salvar ou não durante a varredura é feita no `input_file`.

Descrição da rotina:

São criadas duas strings `s_tl` e `p_tl` que serão utilizadas para nomear os arquivos salvos. O agrupamento destacado em verde no recorte acima são referentes à classe `writeToFile("nome", arquivos)` para salvar arquivos. Um detalhe importante é que arquivos de MPS devem incluir a referência `<MPS>`. Mais informações sobre leitura e escrita de arquivos usando o Itensor podem ser encontradas no link:

http://itensor.org/docs.cgi?vers=cppv3&page=formulas/readwrite_mps

Cálculo dos overlaps

```
#!/ Saídas do cálculo:
//Para entender como U e V estão variando no tempo
cout << "#!U " << lt*del_t << " " << U_t(lt) << "\n"; ///< Valor de U(t) no tempo
cout << "#!V " << lt*del_t << " " << V_t(lt) << "\n"; ///< Valor de V(t) no tempo

//Compute overlap
printfln("Overlap <psi_gs|H|psi_gs> - <psi_evol|H|psi_evol> ");
auto overlap_evol = innerC(psi_Evol,Hv,psi_Evol).real(); ///< Overlap de psi_Evol com H(t)
auto overlap_gs = innerC(psi_GS,H,psi_GS).real(); ///< Overlap de psi_GS com H(t)

cout << "#over_ut " << sqrt(pow(U_t(lt)-U0,2)) << " " << overlap_gs-overlap_evol << "\n";
cout << "#over_vt " << sqrt(pow(V_t(lt)-V0,2)) << " " << overlap_gs-overlap_evol << "\n";
cout << "#over_time " << lt*del_t << " " << overlap_gs-overlap_evol << "\n";
```

Computamos o overlap do estado evoluído (ψ_{Evol}) com o Hamiltoniano instantâneo $H(t)$.
Nesse overlap do estado evoluído (ψ_{Evol}) com o Hamiltoniano instantâneo $H(t)$,
estados de equilíbrio (ψ_{GS}) com o Hamiltoniano instantâneo $H(t)$.

Utilizamos para isso a classe innerC para realizar o sanduíche das funções de onda com o Hamiltoniano instantâneo. Ela funciona da seguinte maneira:

innerC(MPS y, MPO A, MPS x) -> Cplx

Os Hamiltonianos H e H_v são iguais, mas construídos utilizando diferentes arquivos para sites. Essa alteração foi feita para evitar conflitos entre os índices de arquivos de sites salvos em cálculos reiniciados e os índices de sites do cálculo atual (utilizado para avaliar as quantidades relacionadas a ψ_{GS}).