Simulation of the picosecond 2019 paper using the Crank-Nicolson method

Introduction

This script reproduces the results of the Picosecond 2019 paper by numerically solving the time-dependent Schrödinger equation: $i\hbar\frac{\partial\psi}{\partial t}(x,t) = \frac{-\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x,t)\psi(x,t)$ using the Crank-Nicolson method. This script allows to control several parameters that generalizes the simulation performed in the paper. In the entire script, the units of:

- positions are in nm
- times are in ps
- energies are in meV

The fundamental constants $\hbar, m_e, \ldots$ reflect this choice of units and are loaded from a static file.

```
clear sim_output potential grids
```

Simulation parameters

The variables below determine the spatial and time ranges of the simulation.

```
f = 4e9;        % Pumping frequency (in Hz)
T = 1/f * 1e12; % Pumping period (ps)

t_start = 000   % Start time of the simulation
```

```
t_start = 0
```

```
t_end = 015     % End time of the simulation
```

```
t_end = 15
```

```
t_end = T/2;    % End time when using the realistic trajectory

dx = 0.05       % Space precision in nm
```

```
dx = 0.0500
```

```
dt = 0.100      % Time precision in ps
```

```
dt = 0.1000
```

```

x_start = -75  % Position to start the computations from
```

```
x_start = -75
```

```
x_end   = 125   % Position at which to end the computations
```

```
x_end = 125
```

```
% Load custom units
load("meV_nm_ps_units.mat");
m = 0.19*me;    % Electron effective mass
```

Then choose which quantities to compute. The less quantities required, the faster the calculation. This is stored in the `sim_output` structure. A non-zero value enables the associated calculation. The fields are described below:

- `prob_density` -> compute the probability density of the wave function of the electron at each time.
- `average_position` -> compute the average position of the electron at each time. This is useful to analyze later on the period of the oscillations of the electron.
- `energy` -> compute the kinetic, potential and total energy of the wave function in time.
- `occupation_probs` -> perform the scalar product with the expected basis state, giving an indication of the decomposition of the wave function on the basis. This can then be used to compute the occupation probabilities for each energy level and observe their evolution in time.
- `N_states` -> number of energy levels to for which to compute the amplitudes, including the ground state. For instance, if `N_states = 2`, it will be possible to observe the occupation probabilities of the ground state, and the first excited state only.
- `save_workspace` -> if `1`, will save the simulation workspace. Most likely you want this enabled, unless you want to plot the data yourself in Matlab's console (I don't judge ;))

```
sim_output.prob_density = 1;
sim_output.average_position = 1;
sim_output.energy = 1;
sim_output.occupation_probs = 1;
sim_output.N_states = 8;

sim_output.workspace = 1;
```

The potential to use for the simulation is set up using the `potential` structure. The field `type` can take the following values:

- `"harmonic"` -> the potential is the harmonic potential
- `"realistic"` -> the potential is the realistic potential $U(x,t)$
- `"quantum-well"` -> quantum well potential

```
potential.type = "harmonic";

if potential.type == "harmonic"
    % Settings for the harmonic potential
    potential.DeltaE = 1;               % Level spacing of the QD (meV)
    potential.omega = potential.DeltaE / hbar;    % ps^-1
```

```
elseif potential.type == "quantum-well"
    % Settings for the symmetric quantum well potential
    potential.well_width = 50;
    potential.energy_depth = 1e5;

elseif potential.type == "realistic"
    % Get defaults for the realistic potential, and override values if needed
    potential = realistic_potential_default_params(potential);
    potential.f = f;    % Update frequency
end
```

If choosing the harmonic potential, choose the basis to use by setting `basis_choice`. This setting is ignored if the potential used is not harmonic. The following choices are possible:

- `basis_choice = "theory"` -> use the theoretical harmonic oscillator wave functions
- `basis_choice = "numerical"` -> use the harmonic oscillator obtained numerically by solving the associated eigenvalue problem

Note that in the case of both the quantum well and realistic potential, the basis used is loaded from disk. This means that it is necessary to compute these before running the simulation, else it will fail.

```
basis_choice = "theory";
```

The trajectory type is set up using the structure `trajectory`. The `type` field can take the following values:

- `"realistic"` -> normal trajectory based on the minimum of the realistic potential U.
- `"custom-erf"` -> custom erf-based trajectory
- `"static"` -> static (e.g constant, so straight line) trajectory at $x = 0$.
- `"linear"` -> static for $t < t_0$ at $x = 0$, then constant speed until position $x = L$ reached at t_1. Then static again at $x = L$.

```
trajectory.type = "realistic";

if trajectory.type == "realistic"
    % Get defaults for the realistic potential, and override values if needed
    potential = realistic_potential_default_params(potential);
    potential.f = f;    % Update frequency

elseif trajectory.type == "custom-erf"
    % Settings for the erf-based trajectory
    trajectory.t0 = 10;            % Time at which the trajectory of the bottom
abruptly changes
    trajectory.L = 25.0;           % How far to go away from x0
    trajectory.sigma = 1.50;       % How fast the change is
    trajectory.init_position = 00; % Initial position

elseif trajectory.type == "linear"
    % Settings for the linear trajectory
```

```
    trajectory.t0 = 2;
    trajectory.t1 = 4;
    trajectory.x0 = 00;
    trajectory.x1 = 5;
    trajectory.v = (trajectory.x1-trajectory.x0)/(trajectory.t1-trajectory.t0);
end
```

Simulation variables

This section shouldn't need to be modified. It contains various initialization of variables.

```
xgrid = Grid(x_start, x_end, dx);
Nx = xgrid.Npoints;
x = xgrid.Span;

tgrid = Grid(t_start, t_end, dt);
Nt = tgrid.Npoints;
sim_time = t_end-t_start                % Total simulation time in ps
```

```
sim_time = 125.0000
```

```
time_range = (t_start + (0:Nt-1)*dt);   % Simulation time range in ps

qd_min = zeros(Nt, 1);                  % Will contain the position of the minimum
of the QD at each time
```

File settings:

```
grids.xgrid = xgrid;
grids.tgrid = tgrid;
[path, sim_params] = file_manager(grids, potential, trajectory);
```

Initial position:

```
if trajectory.type == "realistic"
    [U_ent_x, U_exit_upper] = realistic_potential_static_part(potential, xgrid);
    U_ent = (potential.V_ent + potential.V_ac * cos(2*pi * f * t_start * 1e-12)) *
U_ent_x;
    U0 = U_ent + U_exit_upper;
    [~, qd_min0] = min(U0);
    x0 = x_start + qd_min0 * dx;
elseif trajectory.type == "custom-erf"
    x0 = custom_qd_trajectory(0, t0, init_position, L, sigma);
elseif trajectory.type == "static"
    x0 = 0;
elseif trajectory.type == "linear"
    x0 = trajectory.x0;
    trajectory.v = (trajectory.x1-trajectory.x0)/(trajectory.t1-trajectory.t0);
end

qd_min(1) = x0;
```

Initial potential value:

```
if potential.type == "harmonic"
    V0 = 1/2 * m * potential.omega^2 * (x - x0).^2;
elseif potential.type == "realistic"
    [U_ent_x, U_exit_upper] = realistic_potential_static_part(potential, xgrid);
    U_ent = (potential.V_ent + potential.V_ac * cos(2*pi * f * t_start * 1e-12)) *
U_ent_x;
    U0 = U_ent + U_exit_upper;
    V0 = U0 * 1e3; % Convert from eV to meV
elseif potential.type == "quantum-well"
    V0 = potential.energy_depth * (abs((x-x0)) > potential.well_width);
end

V = zeros(Nt, Nx);
V(1, :) = V0;
```

Variable initialization:

```
[psi, basis] = init_wave_function(xgrid, Nt, potential, basis_choice, sim_output,
x0, hbar, m);
[prob, average_position, energy, kinetic_energy, potential_energy, wf_amplitudes] =
init_optional_quantities(sim_output, xgrid, Nt, psi, basis, V0, m, hbar);
```

Show final configuration before mail loop:

```
trajectory
```

```
trajectory = struct with fields:
    type: "realistic"
```

```
potential
```

```
potential = struct with fields:
                  type: "harmonic"
                DeltaE: 1
                 omega: 1.5193
         alpha_ent_barr: 0.4900
    alpha_ent_exit_barr: 0.0370
        alpha_exit_barr: 0.4800
     alpha_exit_ent_bar: 0.0520
                 V_ent: -0.7000
                V_exit: -0.7000
                  V_ac: 1.4150
                     f: 4.0000e+09
                 x_ent: 0
                x_exit: 100
                 U_scr: 1
                 L_ent: 100
                L_exit: 100
                 L_scr: 1
```

```
path
```

```
path =
```

```
"../data/DeltaE-1.00/realistic-f-4.0GHz/dx-0.1-xstart-75-xend-125-dt-0.100-tstart-0.000-tend-125.000"
```

Main loop

This shouldn't require modification either. We start by defining some constants related to the coefficients of the matrices

```
I = sparse(eye(Nx));
a = hbar^2/(2*m*dx^2);

c = -1i*dt/(2*hbar)*a;
diag_inf = sparse(c * diag(1*ones(1,Nx-1),1));
diag_sup = sparse(c * diag(1*ones(1,Nx-1),-1));
A_diags = diag_inf + diag_sup;

c = -c;
diag_inf = sparse(c * diag(1*ones(1,Nx-1),1));
diag_sup = sparse(c * diag(1*ones(1,Nx-1),-1));
B_diags = diag_inf + diag_sup;
```

Then comes the main loop:

```
progress = 0
```

```
progress = 0
```

```
tic
for nt = 2 : Nt
    if nt >= (10+progress)*Nt/100
        progress = round(nt/Nt * 100)
    end

    if trajectory.type == "realistic"
        % Time-dependent potential U_ent(x,t)
        U_ent = (potential.V_ent + potential.V_ac * cos(2*pi*f*(t_start+
(nt-1)*dt)*1e-12)) * U_ent_x;

        % Total potential U(x,t)
        U = U_ent + U_exit_upper;

        [~, qd_min_new] = min(U);
        qd_min(nt) = x_start + qd_min_new * dx;
    elseif trajectory.type == "custom-erf"
        % TODO: check if nt or nt-1 below
        qd_min(nt) = custom_qd_trajectory(nt*dt, t0, init_position, trajectory.L,
trajectory.sigma);
    elseif trajectory.type == "static"
        qd_min(nt) = 0;
    elseif trajectory.type == "linear"
        if t_start + (nt-1)*dt <= trajectory.t0
```

```
            qd_min(nt) = trajectory.x0;
        elseif t_start + (nt-1)*dt < trajectory.t1
            qd_min(nt) = trajectory.x0 + trajectory.v*(t_start + (nt-1)*dt-
trajectory.t0);
        else
            qd_min(nt) = trajectory.x1;
        end
    end

    if potential.type == "harmonic"
        V(nt, :) = 1/2 * m * potential.omega^2 * (x - qd_min(nt)).^2;
    elseif potential.type == "realistic"
        V(nt, :) = U*1e3;  % Convert from eV to meV
    elseif potential.type == "quantum-well"
        V(nt, :) = potential.energy_depth * (abs((x-qd_min(nt))) >
potential.well_width);
    end

    % Compute coefficients
    b = 1 + 1i*dt/(2*hbar) * (2*a  + V(nt,:));
    d = 1 - 1i*dt/(2*hbar) * (2*a  + V(nt-1,:));

    % Build matrices
    A = A_diags + b .* I;
    B = B_diags + d .* I;

    % Solve system
    Bpsi = B * psi(:, nt-1);
    psi(:, nt) = A\Bpsi;

    % The wave function has been calculated. Now compute additional
    % quantities as requested
    if sim_output.prob_density
        prob(nt, :) = abs(psi(:,nt)).^2 * dx;
    end

    if sim_output.average_position
        average_position(nt) = trapz(x' .* abs(psi(:,nt)).^2 * dx);
    end

    if sim_output.energy
        [energy(nt), kinetic_energy(nt), potential_energy(nt)] =
wave_packet_energy(psi(:,nt)', V(nt,:), dx, m, hbar);
    end

    if sim_output.occupation_probs
        for n = 1 : sim_output.N_states
            if potential.type == "harmonic"
                wf_amplitudes(nt, n) = trapz(conj(psi(:, nt)') .* ...
```

```
                                        harmonic_oscillator_state(n-1, hbar, m,
potential.omega, qd_min(nt), ...
                                                                  x_start, x_end,
Nx) ...
                                        * dx);
            elseif potential.type == "quantum-well"
                % TODO: PROBABLY NOT WORKING ANYMORE, NEEDS TO BE ADAPTED
                wf_amplitudes(nt, n) = trapz(conj(psi(:, nt)') .* ...
                                        infinite_well_state(n-1, L, qd_min(nt),
x_start, x_end, Nx) ...
                                        * dx);
            end
        end
    end
end
```

```
progress = 10
progress = 20
progress = 30
progress = 40
progress = 50
progress = 60
progress = 70
progress = 80
progress = 90
progress = 100
```

```
psi = psi'; % Switch time/space variables for the plots
toc
```

```
Elapsed time is 164.474244 seconds.
```

```

% Save variables
if sim_output.workspace
    mkdir(path);                                % Create folder for the experiment in
case it doesn't exist
    save(path + "/workspace.mat", '-v7.3');
end
```

```
function [psi, basis] = init_wave_function(xgrid, Nt, potential, basis_choice,
sim_output, x0, hbar, m)
    psi = zeros(xgrid.Npoints, Nt);

    if potential.type == "harmonic"
        if basis_choice == "theory"
            basis = zeros(sim_output.N_states, xgrid.Npoints);
            for n = 1:sim_output.N_states
                basis(n, :) = harmonic_oscillator_state(n-1, hbar, m,
potential.omega, x0, xgrid.Start, xgrid.End, xgrid.Npoints);
            end
            psi(:, 1) = basis(1, :);
        elseif basis_choice == "numerical"
```

```
            % TODO: PROBABLY NOT WORKING ANYMORE, NEEDS TO BE ADAPTED
            basis_file = sprintf("data/realistic-basis-" + "dx-%0.2f-xrange-%d-
%d.mat", ...
                                 xgrid.Step, abs(xgrid.Start), xgrid.End);
            load(basis_file);
            psi(:, 1) = basis(1, :) / 1e9;
        end
    elseif potential.type == "realistic"
        % TODO: PROBABLY NOT WORKING ANYMORE, NEEDS TO BE ADAPTED
        basis_file = sprintf("data/realistic-basis-" + "dx-%0.2f-xrange-%d-%d.mat",
...
                             dx, abs(x_start), x_end);
        load(basis_file);
        psi(:, 1) = basis(1, :);
    elseif potential.type == "quantum-well"
        % TODO: PROBABLY NOT WORKING ANYMORE, NEEDS TO BE ADAPTED
        psi(:, 1) = infinite_well_state(0, L, x0, x_start, x_end, Nx);
    end
end


function [prob, average_position, energy, kinetic_energy, potential_energy,
wf_amplitudes] = init_optional_quantities(sim_output, xgrid, Nt, psi, basis, V0, m,
hbar)
    Nx = xgrid.Npoints;
    x = xgrid.Span;
    dx = xgrid.Step;

    if sim_output.prob_density
        prob = zeros(Nt, Nx);
        prob(1, :) = abs(psi(:, 1)).^2 * dx;
    else
        prob = 0;
    end

    if sim_output.average_position
        average_position = zeros(Nt, 1);
        average_position(1) = trapz(x' .* abs(psi(:,1)).^2 * dx);
    else
        average_position = 0;
    end

    if sim_output.energy
        energy = zeros(Nt, 1);
        kinetic_energy = zeros(Nt, 1);
        potential_energy = zeros(Nt, 1);

        [energy(1), kinetic_energy(1), potential_energy(1)] =
wave_packet_energy(psi(:,1)', V0, dx, m, hbar);
```

```
    else
        energy = 0;
        kinetic_energy = 0;
        potential_energy = 0;
    end


    if sim_output.occupation_probs
        wf_amplitudes = zeros(Nt, sim_output.N_states);

        for n = 1 : sim_output.N_states
            wf_amplitudes(1, n) = trapz(conj(psi(:, 1)') .* basis(n,:) * dx);
        end

    else
        wf_amplitudes = 0;
    end
end
```

Scratchpad