Name: Hongda Li
Class: AMATH 581

1 Intro

This is a organized notes that discussed the theories and major computations problems studied in the class AMATH 581.

The part for Chebyshev Nodes are additional materials presented were not discussed in the class nor the course notes, but its usefulness cannot be overseen and it should be considered as an augmentation to the Spectral Method to a more general context.

2 Quantum Harmonic Oscillator

2.1 Linear

The quantum oscillator can be phrase as a second order differential equation. Writen as:

$$\frac{d^2\phi_n}{dx^2} - (Kx^2 - \epsilon_n)\phi_n = 0$$

We are interested in looking for the Eigenfunctions for the equations.

We expect the solution $\phi_n(x) \to 0$ as $x \to \pm\infty$.

Now we need to convert the behavior of the function over the infinite domain into a boundary conditions for the shooting method.

Let the boundary be symmetric: $[-L, L]$, and then substitute L it into the equation we have the local behavior of the function at the boundary, giving us:

$$\phi_n''(x) - (KL^2 - \epsilon_n)\phi_n(x) = 0$$

The assumption here is that x is very close to L

Solving the linear equation gives:

$$\phi_n(x) = C_1 \exp(\pm\sqrt{KL^2 - \epsilon_n}x)$$

We expect $\phi(x) \to 0$ as $x \to \infty$, which means that:

$$\phi_n(x) = C_1 \exp(-\sqrt{KL^2 - \epsilon_n})$$

Which means that:

$$\phi_n'(x) = -\sqrt{KL^2 - \epsilon_n}\phi_n(x)$$

And this will be the boundary condition at the positive side of the x-axis:

$$\phi_n'(L) + \sqrt{KL^2 - \epsilon_n}\phi_n(L) = 0$$

The boundary condition at the left hand side of the equation is simply:

$$\phi_n'(-L) - \sqrt{KL^2 - \epsilon_n}\phi_n(-L) = 0$$

Note: Take L value that is moderately small for easy implementation for the shooting method, $L = 4$ will be a reasonable quantity.

go to 5.1 for reference on how the shooting method works and matlab codes that implement the shooting scheme for this given system.

Another way of solving it is using the Direct method and phrase it as a matrix Eigenvalue problem.

2.2 Non-Linear

In the case when the function is non-linear:

$$\frac{d^2\phi_n}{dx^2} - (\gamma|\phi_n|^2 + Kx^2 - \epsilon_n)\phi_n = 0$$

And for this part, we assert the same boundary conditions as in part 1.1, however, in the non-linear case, we have the assert an extra constraint that the function must have:

$$\int_{-L}^{L} |\phi_n(x)|dx = 1$$

Because the inner product of the complex function is now sensitive to the initial conditions and the parameters ϵ_n.

Under the case where we want to solve the Quantumn Harmonic Oscillator problem, we need to put the problem into canonical form, giving us a system:

$$y' = \begin{bmatrix} y_2 \\ t^2 - \epsilon y_1 \end{bmatrix}$$

And the parameters we want to tweak is: ϵ, together with the initial conditon A.

To initialize the shooting method, we need to set up:

$$y_0 = \begin{bmatrix} A \\ A\sqrt{KL^2 - \epsilon} \end{bmatrix}$$

Which follows from part 1.1 in the discussion.

For the Linear system, we:

1. Tweak the value ϵ on the left boundary: $x = -L$. **Note:** for the linear system, the solution has the same shape regardless of initial value given to A, and in the end we normalize it to get the probability function.

2. Optimize for the boundary condition on the right: $\phi_n'(L) + \sqrt{KL^2 - \epsilon_n}\phi_n(L) = 0$

Using the shooting method, guess initial condition ϵ_a and ϵ_b such that $\phi_n'(L) + \sqrt{KL^2 - \epsilon_n}\phi_n(L) > 0$ and $\phi_n'(L) + \sqrt{KL^2 - \epsilon_n}\phi_n(L) < 0$ and vice versa, and then by the assumption that the function is continuous wrt to the boundary condition, we will know that the boundary conditions are satisfied for some value in between ϵ_a and ϵ_b.

In addition, the is the caveat of swapping signs for consecutive eigen values, this is called different modes of the eigenvalue.

Let's take a look at the matlab code implementations.

```
import java.util.*;
A = 0.1; L = 4; Epsilon = 1; Xspan = -L: 0.1: L;
Options = odeset("abstol", 1e-13, "reltol", 1e-13);
TOL = 1e-4;                    % Higher than given in HW assignment.

ValidEpsilons = ArrayList();   % Used to stores the valid Epsilons from the shooting method.
EigenFunctions = ArrayList();  % Used to stores the corresponding, normalized eigen functions.

for ModeTrial = 1:5
    EpsilonStep = 2;  % Maximal Distance can move is 2, if oscilates, then it's 1.
    OddSearch = (-1)^(ModeTrial - 1);  % search mode, 1: even functions, -1: odd function
    SearchFlag = 0;  % Search Failed.
    y0 = [A, A*sqrt(L^2 - Epsilon)];

    while 1
```

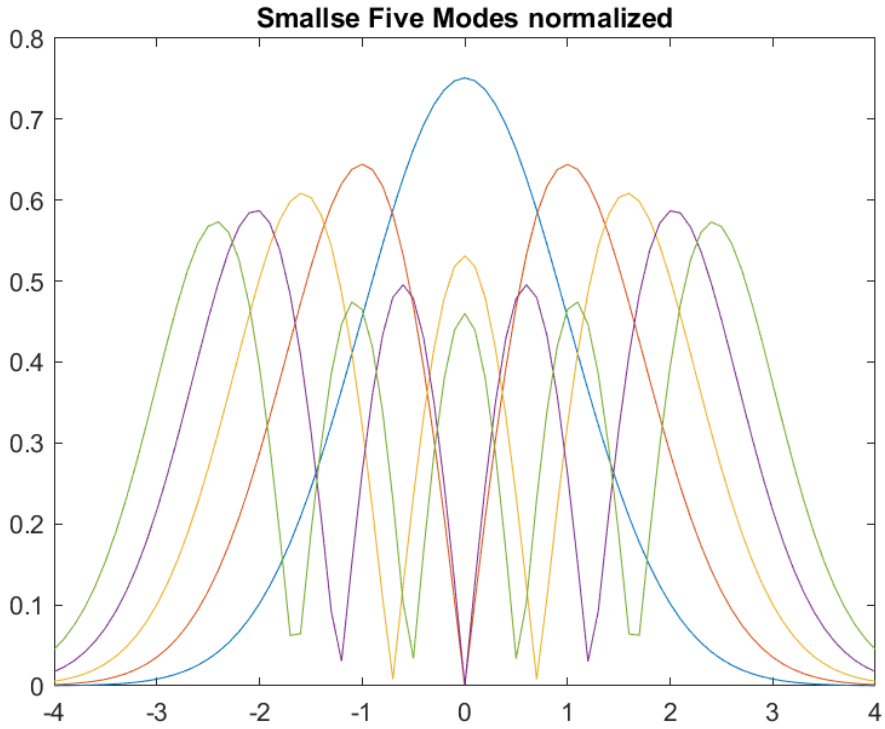
```
        [Xs, Ys] = ode45(@(t, y) Quantum(t, y, Epsilon), Xspan, y0);
        Shoot = Ys(end, 2) + sqrt(L^2 - Epsilon)*Ys(end, 1);
        % Found.
        if abs(Shoot) < TOL
            SearchFlag = 1;
            break
        end
        % Failed.
        if EpsilonStep < 1e-15
           break
        end
        % Try again.
        if Shoot*OddSearch > 0
           Epsilon = Epsilon + EpsilonStep;
        else
           Epsilon = Epsilon - EpsilonStep;
           EpsilonStep = EpsilonStep/2;
        end

    end

    if SearchFlag
        disp(strcat("escaped with BV: ", num2str(Shoot)));
        disp(strcat("escaped with epsilon of: ", num2str(Epsilon)));
        ValidEpsilons.add(Epsilon);
        EigenFunctions.add(Ys(:, 1));
    else
        disp("Search Failed.");
    end
    % increase it by 2 because I know the final answer will be close to
    % this
    Epsilon = ceil(Epsilon) + 2;
end

% Extract Eigen values and normalizing the eigen functions.
Epsilons = zeros(5, 1);
for I = 0: ValidEpsilons.size - 1
    disp(num2str(ValidEpsilons.get(I), '%.8f'));
    Epsilons(I + 1) = ValidEpsilons.get(I);
    EigFx = EigenFunctions.get(I);
    % plot(Xspan, EigFx)
    EigFx = abs(EigFx/sqrt(trapz(Xspan, EigFx.*EigFx)));
    EigenFunctions.set(I, EigFx);
end

A1 = EigenFunctions.get(0);
A2 = EigenFunctions.get(1);
A3 = EigenFunctions.get(2);
A4 = EigenFunctions.get(3);
A5 = EigenFunctions.get(4);
A6 = Epsilons;


function Fxnout = Quantum(t, x, epsilon)
```

```
70      % The linear version of the quantum density function.
71      dy1 = @(t, y, epsilon) y(2);
72      dy2 = @(t, y, epsilon) (t^2 - epsilon)*y(1);
73      Fxnout = [dy1(t, x, epsilon); dy2(t, x, epsilon)];
74  end
```

The top Five Modes of the solutions were found via the shooting method.

1. Mode switching: line 11
2. Solving ODE: line 16
3. Checking boundary conditions: Line 28
4. Decreasing step sizes: Line 31
5. Function Normalization: line: 51

Here is the plot for the first 5 eigenfunctions, normalized found by the code above:



2.3 Nested Shooting Method

The nest shooting method will assert 2 constraints by tweaking 2 parameters. And it's designed to solve the boudnary value problem for the non-linear quantumn harmonic oscillator.

The parameters to tweak is: ϵ, A. The conditions we are asserting are: The boundary conditions and the inner modulus of our solution function.

Here is the psuedo code for the method:

```
while (The solution is not correct):
    Decrease step sizes for both "deltaA" and "DeltaEpsilon"
    while (Function is not Normalized):
```

```
    Do shooting method on parameter: "A"
while (Boundary conditions are not satisfied):
    Do shooting method on parameter: "epsilon"
```

The implementation details are skipped here because it's mostly similar to what we had for the linear cases.

3 Vorticity-Streamfunction Equations

The Vorticity Stream function can be applied to solve some special case of the Naivier Stoke's fluid equations, one of them is the advection-diffusion system of equations.

The stream function is denoted by $\psi(x, y, t)$, and consider the following

$$u = \frac{\partial \psi}{\partial y}$$

$$v = -\frac{\partial \psi}{\partial x}$$

Note that the system is going to be divergence free, meaning that the mass of the particles in the field is conserved.

And the vorticity function: $\omega(x, y, t)$. Which represents the spinning of particle in a tiny unit value.

Then the Advection diffusion equation is summarized as:

$$\frac{\partial w}{\partial t} + [\psi, \omega] = v\nabla^2\omega$$

(Momentum)(eqn 1)

$$\nabla^2\psi = \omega$$

(Vorticity)(eqn 2)

Where the ∇^2 is the Laplacian Operator and the $[\psi, \omega]$ is the corss product in the 2 dimension, which is computed as:

$$[\psi, \omega] = \partial_x\psi\partial_u\omega + \partial_y\psi\partial_x\omega$$

Throughout the discussion of the session, we will focus on solving the problem with **Periodic Boundary Condition**.

3.1 Procedure of solving

To solve the equation, we need to be given an initial conditions: ω_0.

1. Solve for ψ using (eqn 2).
2. Using derivative information to time step and get the next ω using (eqn 1).

The key here is to solve the (eqn 2) efficiently.

Using FFT, or **spectral method**, we will be able to solve it much faster than **direct method**.

3.2 Direct Method

Let D be the second order second differential matrix, with periodic conditions.

Then matrix D is a 3 bands matrix, with coefficient $[1, -2, 1]$ and the band of the matrix will loop back due to periodic boundary conditions.

Then the Laplacian matrix is computed via:

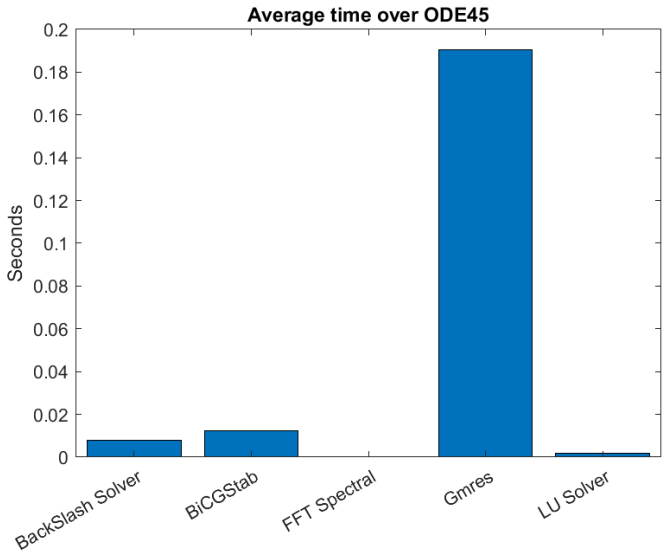
$$D_{\nabla^2} = D \otimes I + I \otimes D$$

And here is a list of linear solver which can be used to solve the system $D_{\nabla^2}\psi = \omega$

1. Backslash
2. Iterative method such as: GMRES, Bicgstab
3. LU Decomposition

For matrices that are relatively small, it's recommended to use LUP decomposition, however when the matrix is big, it's highly recommend to use iterative solver. And for differential matrices that has a diagonal dominant structure.

Here is a benchmark on efficiency of on those ways of solving the Laplacian problem:



Note: The iterative method experience better convergence properties when the initial guess is close, and it's possible to make the ψ_{k-1} sllution as the initial guess for the k th iteration to speed it up.

Note: The Laplacian matrix is not entirely invertible, a constant will be lost, however it will not affect the final solution because the simulation will only needs the derivative of the vorticity function, but this is a problem for the spectral method or direct solver that doesn't handle matrices with null space.

3.3 Spectral Method

The basic of the spectral method is covered in subsection 5.6.

Let's apply the theory to this problem:

Using Fourier Transform:

$$\partial_x^2 \psi + \partial_y^2 \psi = \omega$$

$$\widehat{\omega} = -k_x^2 \widehat{\psi} - k_y^2 \widehat{\psi}$$

$$\widehat{\psi} = -\frac{\widehat{\omega}}{k_x^2 + k_y^2}$$

In practice, take note that because the Laplacian Matrix is not entirely invertible, under this case, the first element of the vector k_x and k_y is zero, and that will make $\psi_{1,1}$ to be an invalid number. And hence we either offset the whole vector off by a 1e-10, or we set the first element of the matrix to be: $1e-6$.

In matlab, the process will be done in the following way:

```
N = params.n;
L = params.l;  % L changed, not the L for LU decomposition anymore
w = reshape(w, N, N);
WFourier = fft2(w);
kx = fftshift((2*pi/L).*(-N/2: (N/2 - 1)));
kx(1) = 1e-6;
ky = kx';
Psi = real(ifft2(-WFourier./(kx.^2 + ky.^2)));
```

```
Psi = reshape(Psi, N*N, 1);
```

Note: Take note that the + operator in matlab will automatically does operator broadcasting, and hence the $k_x^2 + k_y^2$ is in fact a $N \times N$ matrix, and we are dividing the ω, the vorticity function by a matrix, element wise. And then by the end, we vectorized it so it can then be solved with solver like "ODE45".

4 Reaction-Diffusion Systems

The reaction diffusion system is model by the following system of PDEs.

$$u_t = \lambda(A)u - \omega(A)v + D_1\nabla^2 u$$

$$v_t = \omega(A)u - \lambda(A)v + D_2\nabla^2 v$$

Where

$$A = u^2 + v^2 \quad \nabla^2 = \partial_x^2 + \partial_y^2$$

And the terms that make 2 of the agents reacts with each other will be governed by the λ and the ω function, for the sake of HW5 we are looking at these function as:

$$\lambda(A) = 1 - A^2 \quad \omega(A) = -\beta A^2$$

This is a non-linear system of partial differential equations, so it's really fancy however the temporal evolution of the system is explicit.

Take note that the system is a non-linear system, but it can still be solved with the spectral method for Periodic Boundary conditions, but this time we are going to use full spectral method where the time-stepping scheme and system of ODEs will be done in the Fourier domain of the function.

Take note that the system becomes stiff for D_1, D_2 that are not close to zero.

4.1 Problem Setup

Assume that $x, y \in [-L, L]$ discretizing into n points. And D_1, D_2 set to 0.1.

In order to use the Spectral method, we will need to first get the Linear part, and then the non linear part.

$$\nabla^2 u = (K_x \odot K_x + K_y \odot K_y) \odot u$$

Where $\odot$ denotes the Hadamard product, element-wise matrix multiplication product.

Where the matrix K_x, K_y are made by staking the k_x frequencies vector from the Fourier Series.

Rewriting the equation to see the non-linear part of the equation:

$$u_t = (u + (u^2 + v^2)(\beta v - u)) + D_1\nabla^2 u$$

$$v_t = (v - (u^2 + v_2)(\beta u + v)) + D_2\nabla^2 v$$

Generically speaking, suppose that the non linear part for u is simply: $N(u, v) = (u + (u^2 + v^2)(\beta v - u))$, then the procedures for computing the right hand side of u will be the following:

1. Takes in $\widehat{u}, \widehat{v}$ vectorized, in Fourier space.
2. Reshape both into matrix, $\widehat{U}, \widehat{V}$ in Fourier space.
3. Apply "ifft2" to $\widehat{U}, \widehat{V}$ back to the problem domain.
4. Compute $L(U, V)$ by applying the non-linear function to each element of the both matrices.
5. Apply "fft2" on $L(U, V)$, so it becomes $\widehat{L}(U, V)$ and then add the linear part (∇^2 in this case) before vectorizing it and then return this quantity.

Note: Note that, if you are referring the parts that come later in the paper for the Fourier multiplier matrix: K_x, K_y, keep in mind that the example given written for the Spectral Method we have $[-L/2, L/2]$ for the interval, which needs to be re-adapted for this particular problem.

4.2 FFT for Periodic Boundary Condition

Here is an example code, highlighting the major process of evaluating the right hand side of the PDEs equation under Fourier domain.

```
function Argout = RHSFFT(uvvec, params)
    % Given the vector packed with u, v, grids, this function is returning
    % the RHS of the system of ODE, and it will be packaged for the ODEs
    % 45 for evolutions.
    %
    % uvvec:  The vectorized u, v packed together into one big column vectors.
    %       FOURIER DOMAIN.
    % params: An instance of ProblemParameters, containing all the
    %       parameters need to do the computations.


    [UFMtx, VFMtx] = params.VectorUnpack(uvvec);
    A2Tdomain      = Asquared(UFMtx, VFMtx);

    dudthat = DuDtHat(UFMtx, VFMtx);
    dvdthat = DvDtHat(UFMtx, VFMtx);
    Argout  = params.VectorPack(dudthat, dvdthat);


    function Argout = DuDtHat(uFMtx, vFMtx) % Foutier domain, Matrix.
        U      = ifft2(uFMtx);
        V      = ifft2(vFMtx);
        Argout = lambda().*U - omega().*V;
        Argout = fft2(Argout);              % To Fourier.
        Argout = Argout + params.D1.*Laplacian(uFMtx);
    end

    function Argout = DvDtHat(uFMtx, vFMtx) % Foutier Domain, Matrix.
        U      = ifft2(uFMtx);
        V      = ifft2(vFMtx);
        Argout = omega().*U + lambda().*V;
        Argout = fft2(Argout);
        Argout = Argout + params.D2.*Laplacian(vFMtx);
    end

    function Argout = lambda()          % problem domain out, matrix form
        Argout = 1 - A2Tdomain;
    end

    function Argout = omega()           % problem domain out, matrix form
        Argout = -params.beta.*A2Tdomain;
    end

    function Argout = Laplacian(xfMtx) % problem domain out, matrix form
        Argout = params.Laplacian.*xfMtx;
    end
end


function Argout = Asquared(uFourier, vFourier) % IN: FOURIER DOMAIN MATRIX FORM
```

```
51
52      u = ifft2(uFourier);
53      v = ifft2(vFourier);
54
55      % Fact: u, v should be mostly reals, because we are under problem
56      % domain
57
58      Argout = u.^2 + v.^2;
59
60  end % OUT: TIME DOMAIN MATRIX FORM
```

Everything is modular and packed into different functions, so we know nice and clear, what is happening exactly.

4.3 Chebyshev for Dirichlet Boundary Conditions

This is achieved via looking for the chebyshev differential matrix to represent the laplacian. Let the chebyshev Differential matrix be D, then the Laplacian operation matrix will be given by:

$$D_{\nabla^2} = \frac{1}{L^2}(D \otimes I + I \otimes D)$$

However, we need to take care of the boundary conditions. For simplicity, we can set the boundary conditions to be $u(\pm L, y), u(x, \pm L) = 0$.

This is achieve by setting the first/last row of the matrix D to be zero before we commpute the D_{∇^2} with kronecker product.

Here is the matlab implementations for the right hand side of ODEs when using the Chebyshev method:

```
1  function [DudtOverDvdt] = RHSCheb(uv, params)
2      % Using the chebyshev matrix to compute the right hand side, and it's
3      % Dirichlet boundary conditions.
4
5      u      = uv(1: (params.n + 1)^2);       % Vectorized u
6      v      = uv((params.n + 1)^2 + 1: end); % Vectorized v
7      A2     = u.^2 + v.^2;
8      Lambda = 1 - A2;
9      Omega  = -params.beta.*A2;
10     Laplac = params.chebLaplacian;
11
12     Ut = F1(u, v);
13     Vt = F2(u, v);
14
15     DudtOverDvdt = [Ut; Vt];
16
17     function Ut = F1(u, v)
18         Ut = Lambda.*u - Omega.*v + params.D1.*Laplac*u;
19     end
20
21     function Vt = F2(u, v)
22         Vt = Omega.*u + Lambda.*v + params.D2.*Laplac*v;
23     end
24
25 end
```

Note: When visualizing the solution, please use the Chebyshev Nodes or else it's going to be distorted.

5 Theory and Algorithms

5.1 Shooting Method

1.1em The shooting method is a general way of solving boundary value problem for single variable, function, applied to the second order derivative.

Generically, assuming that we were given the equation:

$$\frac{d^2y}{dt^2} = f\left(\frac{dy}{dt}, y, t\right) \quad t \in [a, b]$$

And the boundary condition is in the form of:

$$\alpha_1 y(a) + \beta_1 \frac{dy}{dt}(a) = \gamma_1$$

$$\alpha_2 y(b) + \beta_2 \frac{dy}{dt}(b) = \gamma_2$$

The problem is phrased as an optimization problem, assuming a function $S(\alpha_1, \beta_1)$ which takes in initial condition at $x = -L$ and then returns difference for the boundary at $x = L$

Then the initial condition α_1, $_1$ is essentially give by:

$$\underset{\alpha_1, \beta_1}{\operatorname{argmin}}(S(\alpha_1, \beta_1))$$

And the solution to the optimization problem solves the boundary value parblem.

However, in practice, the shooting method is implemented via the bisection method. Take notes that any generic optimization algorithm will still work, for example we can use the built in matlab “fzero” command for looking for the solution that satisfied the boundary conditions.

5.2 Direct Method

The direct method is a way of discretizing the domain of the function and use a vector to represent function. This is useful because the function is a vector and the differential equation will be a linear system, which can be solve directly using linear algebra.

The Direct method applies to BVP with the following form:

$$y'' = p(t)y' + q(t)y + r(t)$$

PDEs are also ok but it requires more sophisticated way of constructing the discretized differential operators.

The idea here is to use finite difference method to look for a linear transformation for the differential operator. All the discretization method must be at the same order of accuracy.

In this generic case, we will have:

$$y'(t) = \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2)$$

$$y''(t) = \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2}$$

Take notice that this operations can be represented by a 3 band matrix for both operator, when the boundary conditions are not yet involved, the matrix is like:

$$2\Delta t\partial_x = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \cdots \\ -1 & 0 & 1 & 0 & 0 & \cdots \\ 0 & -1 & 0 & 1 & 0 & \cdots \\ 0 & 0 & -1 & 0 & 1 & \cdots \\ \vdots & & & & & \\ \cdots & 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

And this is the matrix that takes the first derivative with finite difference on a given discretized domain of a function. For the second order second deritive, we just ue $[1, -2, 1]$ as the coefficients on the 3 bands of the matrix, which looks like:

$$\Delta^2 t\partial_x = \begin{bmatrix} 2 & 1 & 0 & 0 & 0 & \cdots \\ -1 & 2 & 1 & 0 & 0 & \cdots \\ 0 & -1 & 2 & 1 & 0 & \cdots \\ 0 & 0 & -1 & 2 & 1 & \cdots \\ \vdots & & & & & \\ \cdots & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

5.3 Direct Method: Robin Boundary

Let's consider a generic, linear boundary conditions:

$$\alpha_1 y(t_0) + \beta_1 y'(t_0) = \gamma_1$$

$$\alpha_2 y(t_N) + \beta_2 y'(t_N) = \gamma_2$$

Using Finite Difference method on the boundary conditions, we have:

$$\alpha_1 y(t) + \beta_1 \frac{y(t+2\Delta t) - 4y(t+\Delta t) + 3y(t)}{2\Delta t} = \gamma_1$$

And

$$\alpha_1 y(t) + \beta_1 \frac{-y(t+2\Delta t) + 4y(t+\Delta t) - 3y(t)}{2\Delta t} = \gamma_1$$

Replacing the variables on the first boundary equation: let $t + 2\Delta t$ to be t_1, $t + \Delta t$ be t_1 and t be t_0, which represents the first 3 points close to the left boundary of the discretized domain of the function.

Do the same thing for the right boundary, and then we have a new system of equations involving the boundary information:

$$\alpha_1 y(t_0) + \beta_1 \frac{y(t_2) - 4y(t_1) + 3y(t_0)}{2\Delta t} = \gamma_1$$

And right boundary is gonna be:

$$\alpha_1 y(t) + \beta_1 \frac{-y(t_N) + 4y(t_{N-1}) - 3y(t_{N-2})}{2\Delta t} = \gamma_1$$

These 2 equations are going to be the first and the last row of the discretized differential matrix, and the constant on the right hand side is the boundary conditions which will appear in the right hand side of the matrix equation as well.

Write a matrix to represent the linear operator: $\partial_t^2 + \partial_t$; and the linear term $q(t)y$ is just a diagonal matrix with it's diagonal equals to $q(t_i)y(t_i)$. Sum all above linear operators together and name the matrix A.

The vector b to include the boundary conditions and the right hand side of the equations, which gives us:

$$b = \begin{bmatrix} \alpha \\ r(t_1)\Delta t^2 \\ t(t_2)\Delta t \\ \vdots \\ r(t_N)\Delta t^2 \\ \beta \end{bmatrix}$$

And hence solving the system of the equation numericall equations to solving the system:

$$At = b$$

And take note that in this case the matrix $(N+1) \times (N+1)$ when it's not a periodic boundary conditions.

5.4 Direct Method: Higher Dimension

Direct method can be applied to higher dimension, using the Kronecker Product and vectorization of the high dimensional domain, we will be able to put a high dimension problem into a matrix vector system.

Suppose that we were given a discrete partial differential operators ∂_x fora 2 dimensional problem. And suppose that we pack the domain of the by discretizing along the columns (same x value) and then y value.

More specifically, suppose that $\psi(x_i, y_j) = \psi_{i,j}$, and the grid is uniformly discretized, then the vectorized domain can be described as:

$$\vec{\psi} = \begin{bmatrix} \psi_{1,1:N} \\ \psi_{2,1:N} \\ \vdots \\ \psi_{N,1:N} \end{bmatrix}$$

Then our ∂_x differential operator for 2d is simply:

$$\frac{d}{dx} \otimes I$$

And conveniently, we will have the operator for ∂_y as:

$$I \otimes \frac{d}{dx}$$

Note: This is very general and it encapsulate both the boundary conditions and it's applicable to infinite dimension. As long as the discretized differential operator for One dimension contains the correct boundary conditions, then the kron product will retain the boundary conditions into higher dimension.

5.5 Spectral Method

The spectral method is based on DFT, it has the following quirks:

1. Operations is $\mathcal{O}(N\log(N))$, for uniformly discretized spatial dimension. Other topology are not applicable for DFT.
2. It's only suitable for Periodic Conditions, if not the boundary will be look really ugly and there will be oscillations. (And in that case please consult prof Kutz book on Scientific Computing, where 2 pinned boundary conditions is discussed for the spectral methods)
3. Discretization has to be 2^n, this is best discretization for the FFT algorithm to work with.
4. The order of accuracy of the method is exponential, so it's pretty insane.

The spectral method exploit the properties that the sin and cos basis are orthogonal, so we can transform back and forth.

It also exploit the linear properties of Fourier Transform. More specifically:

$$\widehat{f^{(n)}(x)} = (ik)^n \widehat{f(x)}$$

And under the case of Discrete Fourier transform, the transformation vector needs to be transform via the "fftshift" command in matlab, this is necessary because the Fourier Series is a Bi-infinite sum that sum up from the negative infinity to positive infinity, from high frequencies to low frequencies and then to high frequency, but the DFT matrix is starting with the lower frequencies and then goes to the high frequencies when taking the inner product with the basis.

Here is an example code of taking derivative using the "fft" command in matlab:

```
L  = 5;
N  = 2^8;
x2 = linspace(-L/2, L/2, N + 1);  % Query points.
x  = x2(1:N);                     % Trim off the last point because of periodic conditions.

u = exp(-x.^2)                    % cos(x).*exp(-x.^2/25);
U_Fourier = fft(u);

k = (2*pi/L)*(-N/2: N/2 - 1);  % The lower infinite and upper inifity on the fourier transform.
k = fftshift(k);               % Must shift this or it's not working.

figure(1);
plot(x, u); hold on;

% Taking derivative on the Fourier Space it's gonna be like:

dU_F  = i.*k.*U_Fourier;
dU2_F = (i.*k).^2.*U_Fourier;
dU    = ifft(U_Fourier);
plot(x, real(dU));
plot(x, real(ifft(dU_F)));
plot(x, real(ifft(dU2_F)));
```

We usually denote the vector the does the linear transformation using k_x for the DFT transform.

Note The domain is $[-L/2; L/2]$, please adapt this condition accordingly for different types of domain on the real lines.

5.6 Spectral Method: Full Spectral

Full spectral method refers to solving a system of PDEs entirely under the Fourier Domain.
Here is the procedure for solving a given system with Full Spectral Method.
Let's start with an input $\widehat{u}$, denoting a function represented in the Fourier Domain. Then:

1. Convert $\widehat{u}$ back to Function Domain, using "ifft".
2. Plug the function into the Right hand side of u_t, and the function should be non-linear.
3. Get the result out, say it's $N(u)$, and then we performs a "fft" to take the output of the RHS to fourier space.
4. Vectorize the solution so it can be solved using solvers like: "ODE45".

5.7 Spectral Method: Other Boundaries

The spectral method can also be applied for:

1. Pinned Boundary condition: $f(0) = f(L) = 0$
2. no-flux Boundary conditions: $f'(0) = f'(L) = 0$

Given any function on the domain $[0, -L]$, we have the choice to augment the function at the interval $[-L, 0]$ either by setting $f(x) = f(-x)$, which corresponds to an even function or $f(x) = -f(-x)$ which corresponds to an odd function.

The former is the Discrete Sine Cosine transform and the later is the Discrete Cosine transform.

And in our case, different type of transform uses an even and odd function to interpolate the solution. And both type of interpolation can be computed using the FFT algorithm.

Note: In matlab, the commands for the above transform exists and they are "dst", "dct".

Suppose that we were given the function to simulate over the interval: $[0, L]$, and we want to apply the Fourier Sine, Cosine transform.

For cosine transform, we mirror image around the y-axis and then apply the Fourier transform. The Fourier series is:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n \exp\left(\frac{n\pi x}{L}\right)$$

Where the c_n is just the Fourier Coefficients.

And take notice that because we mirror the function, then the function will be an even function, therefore, it only has projection on the the cos component, which means that it's:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n \cos\left(\frac{n\pi x}{L}\right)$$

The cos it able to model any type of boundary conditions for the function $f(x)$, and the derivative will be given as:

$$f'(x) = \sum_{n=-\infty}^{\infty} -c_n \left(\frac{n\pi}{L}\right) \sin\left(\frac{n\pi x}{L}\right)$$

And take not that, for any $n \in \mathbb{Z}$, we will have that $x = L, 0$ on the boundary conditions and then it will mean that we are summing up $\sin(0)$ or $\sin(n\pi)$, and this means that:

$$f'(L) = f'(0) = 0$$

And this is the **Neuamnn Boundary Conditions**, or the no flux boundary conditions.

Similarly, if we interpolate the boundary using the sin function by reflecting the function from the $[0, L]$ to the $[-L, 0]$ domain, then:

$$f(L) = f(0)$$

And this is the **Pinned Boundary Conditions**.

5.8 Chebyshev Nodes: Chebyshev Differentiation Matrix

In this part we will be focusing on how the chebyshev differential matrix is made using the idea of polynomial interpolations and Lagrange Polynomials.

The chebyshev Nodes are the collection of points what are exactly at the roots of the chebyshev polynomials, using these set of points to interpolate an analytical function minimizes the Runge Phenomena.

Chebyshev Nodes are defined as:

$$x_j = \cos(j\pi/N) \quad \forall\, j = 0, 1, 2 \cdots N$$

And it's over the interval: $[-1, 1]$

A polynomial interpolation scheme using the Lagrange basis is:

$$L(x) := \sum_{j=0}^{k} y_j \left(\prod_{\substack{m=0 \\ m \neq j}}^{k} \frac{x - x_m}{x_j - x_m} \right)$$

Which is a linear function wrt to the points of interpolations, y_j, and in this case y_j is the function value at x_j the chebyshev nodes. Take notice that there is nothing special about using the chebyshev nodes here, the reason chebyshev nodes is used it's because that it has higher accuracy. And it's related chebyshev basis which can be exploit via the FFT algorithm.

Discretizing the given function along the points of interpolation is essentially a linear system, which can be written as the following:

$$\begin{bmatrix} L(x_0) \\ L(x_1) \\ L(x_2) \vdots L(x_N) \end{bmatrix}$$

And hence taking the derivative of this will still give use a linear system using the interpolating points:

$$\begin{bmatrix} L'(x_0) \\ L'(x_1) \\ L'(x_2) \\ \vdots \\ L'(x_N) \end{bmatrix}$$

The entries of the cheb matrix is literally given from the derivative of the interpolative polynomials, by which I mean that:

$$D_{i,j} = L'(x_i)$$

Ok... So then we have to use the derivative from the Lagrange Polynomials to help us out here.

Picking up from the last part, we have the following quantities:

$$l_j(x) = \prod_{\substack{m=0 \\ m \neq j}}^{k} \frac{x_j - x_m}{x_j - x_m} = 1$$

and

$$p_j(x) = \frac{1}{a_j} \prod_{\substack{k=0 \\ k \neq j}}^{N} (x - x_k)$$

And:

$$a_j = \prod_{\substack{k=0 \\ k \neq j}}^{N} (x_j - x_k)$$

And the derivative of $p'(x)$ is given by:

$$p'_j(x) = p_j(x) \sum_{\substack{k=0 \\ k \neq j}}^{N} \frac{1}{x - x_k}$$

Then this means that the interpolated function has derivative of:

$$\sum_{j=0}^{k} f_j p'_j(x) = \sum_{j=0}^{k} f_j \left(p_j(x) \sum_{k=0, k \neq j}^{N} (x - x_k)^{-1} \right)$$

Convert the inner $p_j(x)$ to the finite product form:

$$\sum_{j=0}^{k} f_j p'_j(x) = \sum_{j=0}^{k} f_j \left(\frac{1}{a_j} \prod_{\substack{k=0 \\ k \neq j}}^{N} (x - x_k) \sum_{k=0, k \neq j}^{N} (x - x_k)^{-1} \right)$$

Move out the constants and make the running index **independent**"

$$\sum_{j=0}^{k} f_j p'_j(x) = \sum_{j=0}^{k} \frac{f_j}{a_j} \left(\prod_{\substack{k=0 \\ k \neq j}}^{N} (x - x_k) \sum_{l=0, l \neq j}^{N} (x - x_k)^{-1} \right)$$

Let's fix the value of j, and then we consider the inner sum:

$$\left(\prod_{\substack{k=0 \\ k \neq j}}^{N} (x - x_k) \sum_{l=0, l \neq j}^{N} (x - x_k)^{-1} \right)$$

Which can be multiplied in and obtain:

$$\sum_{\substack{l=0 \\ l \neq j}}^{N} \left(\prod_{\substack{k=0 \\ k \neq j \\ k \neq l}}^{N} (x - x_k) \right)$$

Now, we set $x = x_i$, assuming that $i \neq j$
Then take notice that, $k \neq j$ implies that $\exists\, k$ such that $k = i$.
Then the inner produce will be zero for all $l \neq i$
And then the only non zero terms is when $l = i$.
And then we have:

$$\prod_{\substack{k=0 \\ k \neq i,j}}^{N} (x_i - x_k)$$

Therefore, the coefficient of f_j for the term $L(x_i)$ where $i \neq j$, we have:

$$\frac{f_j}{a_j} \prod_{\substack{k=0 \\ k \neq i,j}}^{N} (x_i - x_k) \tag{1}$$

And take notice that, multiplying both numerator and denominator by $(x_i - x_j)$, we will have:

$$D_{i,j} = \frac{f_j a_i}{a_j(x_i - x_j)}$$

However, go back to the very beginning and assume that the value x_i happens to have $i = j$, then we have:

$$p'_j(x_j) = p_j(x_j) \sum_{\substack{k=0 \\ k \neq j}}^{N} \frac{1}{x_j - x_k}$$

And $p_j(x_j) = l_j(x_j)$, and then that means:

$$p'_j(x_j) = \sum_{\substack{k=0 \\ k \neq j}}^{N} \frac{1}{x_j - x_k} = D_{j,j}$$

The chebyshev differential matrix is explained well in Trefethen's Work titled "Spectral Method in Matlab", and after simplifying equation (1) we will be getting a much much concise description for the matrix:

$$\begin{gathered}
(D_N)_{00} = \tfrac{2N^2+1}{6}, \quad (D_N)_{NN} = -\tfrac{2N^2+1}{6} \\
(D_N)_{jj} = \tfrac{-x_j}{2\left(1-x_j^2\right)}, \quad j = 1, \ldots, N-1 \\
(D_N)_{ij} = \tfrac{c_i}{c_j} \tfrac{(-1)^{i+j}}{(x_i - x_j)}, \quad i \neq j, \quad i, j = 1, \ldots, N-1
\end{gathered}$$

where

$$c_i = \begin{cases} 2 & i = 0 \text{ or } N \\ 1 & \text{otherwise} \end{cases}$$

However at this point take notice of the following unresolved problem:

1. The matrix is dense and its computations requires $\mathcal{O}(N^2)$.
2. What boundary does it supports on a non-uniformly discretized grid?
3. We need to be careful when extending the method for intervals such as $[-L, L]$, we need to apply a multiplier $1/L$ whenever we use the Chebyshev differential matrix.

5.9 Chebyshev Nodes: Boundary Conditions

The chebyshev method is amazing for Dirichlet Boundary conditions.

For pinned boundary conditions where $u(\pm L) = 0$, we will remove the parts in the D matrix where the boundary points are associated with and set them all to zeros.

This is achieved via forcing the value of the boundary points to be some kind of value, in this case we set $x_N, X_0 = 0$, and that will set all the boundary of the Chebyshev matrix to be zeros.

$$\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & D_{N-1 \times N-1} & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

For General boundary conditions, suppose that we have: $u(-1) = -2, u(1) = 2$, and an equation:

$$\partial_x^2 u = 0$$

Discretizing it we will have:

$$D^2 u = \mathbf{0}$$

And take note that, now we need to do the boundary surgery:

$$\begin{bmatrix} D^2_{0,0} & \cdots & D^2_{N+1,N+1} \\ \vdots & \widetilde{D}^2 & \vdots \\ D^2_{N+1,0} & \cdots & D^2_{N+1,N+1} \end{bmatrix} u = \mathbf{0}$$

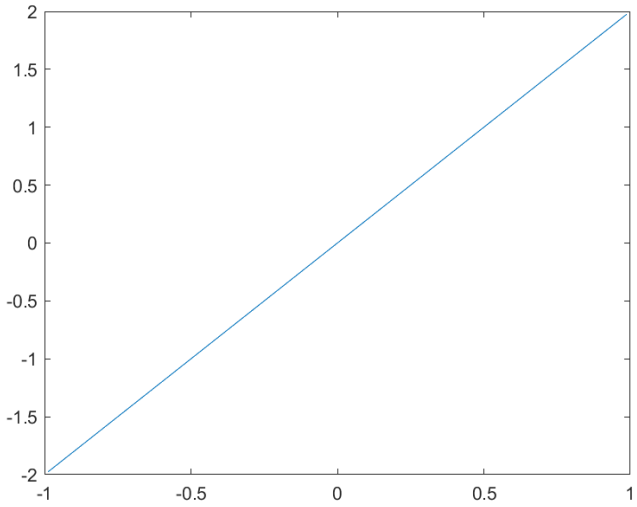
Where the matrix $\widetilde{D}^2$ denotes the interior parts of the D^2 matrix.

The key here is that, by asserting that: $u_0 = -2$ and $u_{N+1} = 2$, we will be able to move the columns of the matrix D^2 to the right hand side where it becomes a vector b, a $N - 1 \times 1$ vector.

Here is some matlab example codes of doing these:

```
[D, chebPoints] = cheb(20);
D2 = D^2;
b = -2.*D2(:, 1) + 2.*D2(:, end);
b = b(2: end - 1);
D2Tilde = D2(2: end - 1, 2: end - 1);
plot(chebPoints(2: end - 1), (D2Tilde)\b);
```

Where the "cheb" function is written in Trefethen's Book "Spectral Method in Matlab", and this will make the plot of the function:



Note: The cheb method can be easily extended to more complex boundary conditions, all we need is to do the surgery on the matrix according to our boundary conditions when we need it.

For example the boundary conditions can be: $u(-L) = \alpha(t), u(L) = \beta(t)$, then in this case, we will need to move the columns around with the Forced Boundary conditions at each moment in time when evaluate the right hand side.

5.10 Chebyshev Nodes: Chebyshev Using FFT

Chebyshev is just another spectral method under the hood, we have seem that it as flexible as the Direct Method with Finite Difference, and now it's time to see how it FFT can be applied to the cheybshev matrix, and how it then can be used to speed up the computations of derivative.

Let's define the chebyshev polynomials, which is a basically a polynomials that have roots for all the chebyshev nodes, of the definitions is:

$$T_n(x) := \Re(z^n) = \frac{1}{2}(z^n + z^{-n}) = \cos(n\theta)$$

Where z is the root of unity.

Beautiful, we know let's focus on the theoretical parts and assuming everything ins nice and smooth, before we move to the more discrete cases.

Let's say we interplated a function with the Chebyshev Polynomial:

$$p(x) = \sum_{n=0}^{\infty} a_n T_n(x)$$

Using the properties of the Chebyshev Polynomials, we have:

$$p(\cos(\theta)) = \sum_{n=0}^{\infty} a_n \cos(n\theta) \quad \text{Where: } x = \cos(\theta)$$

And we have a new interpolative function, name it: $P(\theta)$, then we have:

$$P(\theta) = \sum_{n=0}^{\infty} a_n \cos(n\theta)$$

And taking derivative wrt to x we use chain rule and get:

$$\frac{dP(\theta)}{dx} = \frac{dP(\theta)}{d\theta}\frac{-1}{\sqrt{1-x^2}}$$

Take note that, the function $P(\theta)$ is way faster to get, using FFT, or discrete Cosine Transform to determine.

Take the observation that, as θ goes from $-\pi$ to 0, $\cos(\theta)$ goes from -1 to 1, and that is the whole domain of interpolations on the real functions. As θ goes from 0 to theta π, $\cos(\theta)$ goes from 1 to -1.

$$P(\arccos(x)) = p(x) \quad P(\theta) = p(\cos(\theta))$$

And if θ is uniformly distributed along $[-\pi, \pi]$, then $\cos(\theta)$ will produces all the **Chebyshev Nodes**. And this is the part where we have the way of getting $P(\theta)$ directly from the function value at Chebyshev nodes!

Here is the Algorithm for getting all the interior points using the FFT algorithm

1. Given a set of values, v_i where i goes from 0 to N, these are the function value interpolated at the chebyshev nodes.

2. Reorder the $N+1$ data points on the $[-1, 1]$ in this way:

$$V^T = [v_0\ v_1\ \cdots\ v_N\ v_{N-1}\ v_{N-2} \cdots\ v_1]$$

3. Use FFT algorithm to interpolate the vector V, get $\widehat{V}$.

4. Take the derivative on the vector using the k_x vector, name the vector after derivative W.

5. Then for all the interior points, the derivative value is given by:

$$w_j = \frac{-W_j}{\sqrt{1-x_j^2}} \quad \forall 1 \leq j \leq N$$

6. And for the boundary points, we will have;

$$w_0 = \frac{1}{2\pi}\left(\frac{N^2\widehat{V}_N}{2} + \sum_{n=1}^{N-1} n^2\widehat{V}_n\right)$$

$$w_n = \frac{1}{2\pi}\left(\frac{(-1)^{N+1}N^2\widehat{V}_n}{2} + \sum_{n=1}^{N-1}(-1)^{n+1}n^2\widehat{V}_n\right)$$

For a thorogh implementation of the algorithm, please refer to program 18 in Trefethen's Book "Spectral Method in Matlab"