NAnPack

Release 1.0.0-alpha4

Dr. Vishal Sharma

Feb 28, 2021

CONTENTS:

Welcome to NAnpack tutorials page. This is the space where you can find the tutorials on how to use this package.

CHAPTER

ONE

INSTALLATION (V1.0.0-ALPHA4)

1.1 For Windows OS

1.1.1 I. Requirements

The package requires Python 3, which may be downloaded from the Python homepage. It comes with the integrated environment IDLE and Python Shell. The other popular development environment recommended for Python is Jupyter Notebook which is also open-source.

1.1.2 II. Approach 1 - Installing NAnPack from source

One straightforward way of installing without having to use Git is downloading zip files from the GitHub repository.

1. Visit GitHub project page, link here.
2. Download ZIP in the target directory `/path/to/myproject` and unzip the contents.
3. On your terminal/command window, change the directory to the root of the unzipped NAnPack installation directory where "setup.py" is located `cd /path/to/myproject`.
4. Install nanpack using the following command.`python setup.py install`

This process will ensure that that you have downloaded the required configuration files located in the `./input/` folder.

1.1.3 III. Approach 2 - Installing NAnPack using PIP

NAnPack is uploaded on Python Package Index (PyPI) repository and thus it can be easily installed by entering the following on your terminal:

```
pip install nanpack
```

To get the configuration files, you may have to dig into the directory where all python packages are installed and copy the input folder to the target directory for your projects.

If you don't have PIP installed, first read 'Check PIP' section and then continue from here.

1.1.4 IV. Check Installation

To check correct installation of the package, run the following tests on your command window/terminal

1. Test nanpack installation - `python -m nanpack.tests.test_nanpackinstall`
2. Test required third-party packages - `python -m nanpack.tests.test_thirdpartyinstalls`. If this test fails, proceed to the section V.
3. Run an example case to test everything is working - `python -m nanpack.tests.test_run`.

The detailed outputs from these tests can be *found here*.

If Test#2 is passed, skip the below sections V and VI.

1.1.5 V. Installing other Python packages

Following are the required additional third-party packages to ensure correct functioning of NAnPack - NumPy and Matplotlib.

Whether or not these packages are installed on your system can be checked by entering on the command window:

```
pip show <package-name>
```

If they are not already installed, type the following in the command window:

```
pip install numpy
pip install matplotlib
```

1.1.6 VI. Check PIP

After you have downloaded the Python environment, we will use PIP to install packages/modules. PIP is the package manager for Python modules which is included by default with Python 3.4 or above. First check whether PIP is installed correctly by typing the following command in the command window and enter

```
C:\Users>pip --version
```

The output should be similar to as shown below

```
<pip 20.2.4 from c:\users\owner\appdata\local\programs\python\python37-32\lib\site-
↪packages\pip (python 3.7)>
```

If it does not work, check that the Python directory is included in your system environment PATH variable or re-install Python or try installing PIP.

[ ]:
```
# Document Author: Dr. Vishal Sharma
# Author email: sharma_vishal14@hotmail.com
# License: MIT
# This tutorial is applicable for NAnPack version 1.0.0-alpha4
```

CHAPTER

TWO

RUNNING TESTS

Users are advised to follow the instructions given here to run the required tests on Jupyter Notebook after installing the package.

Enter these commands on terminal or Jupyter Notebook to run the tests.

Note:
On Windows command line remove "%run" from the commands given below. "%run" is the magic IPython command used in Jupyter Notebook.

2.1 Test # 1

First test is to verify whether the nanpack is installed correctly on your system. This is performed by running "test_nanpack.py" file in the `tests` folder of your project root. Enter the command shown in cell 2 (replace with correct project path) of this notebook and verify that the test is completed successfully as shown in the ouput below.

[1]:
```
%run -m nanpack.tests.test_nanpackinstall
```

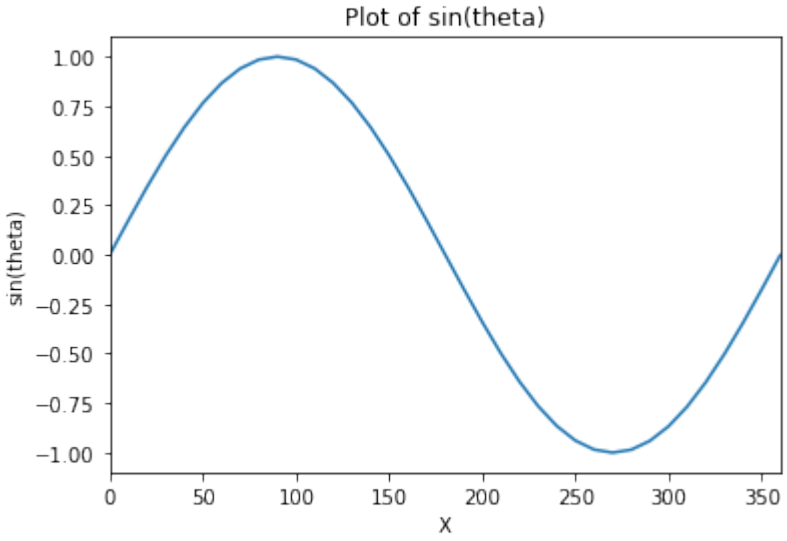
```
NAnPack package test SUCCESS.
```

2.2 Test # 2

Next test is to verify whether the required third party packages- NumPy and matplotlib are installed correctly on your system. Run script "test_thirdpartyinstalls.py" by entering the command shown in cell 3 (replace with correct project path) and verify that the test is completed successfully as shown in the ouput below.

[2]:
```
%run -m nanpack.tests.test_thirdpartyinstalls
```

```
Close plot to continue testing.
```



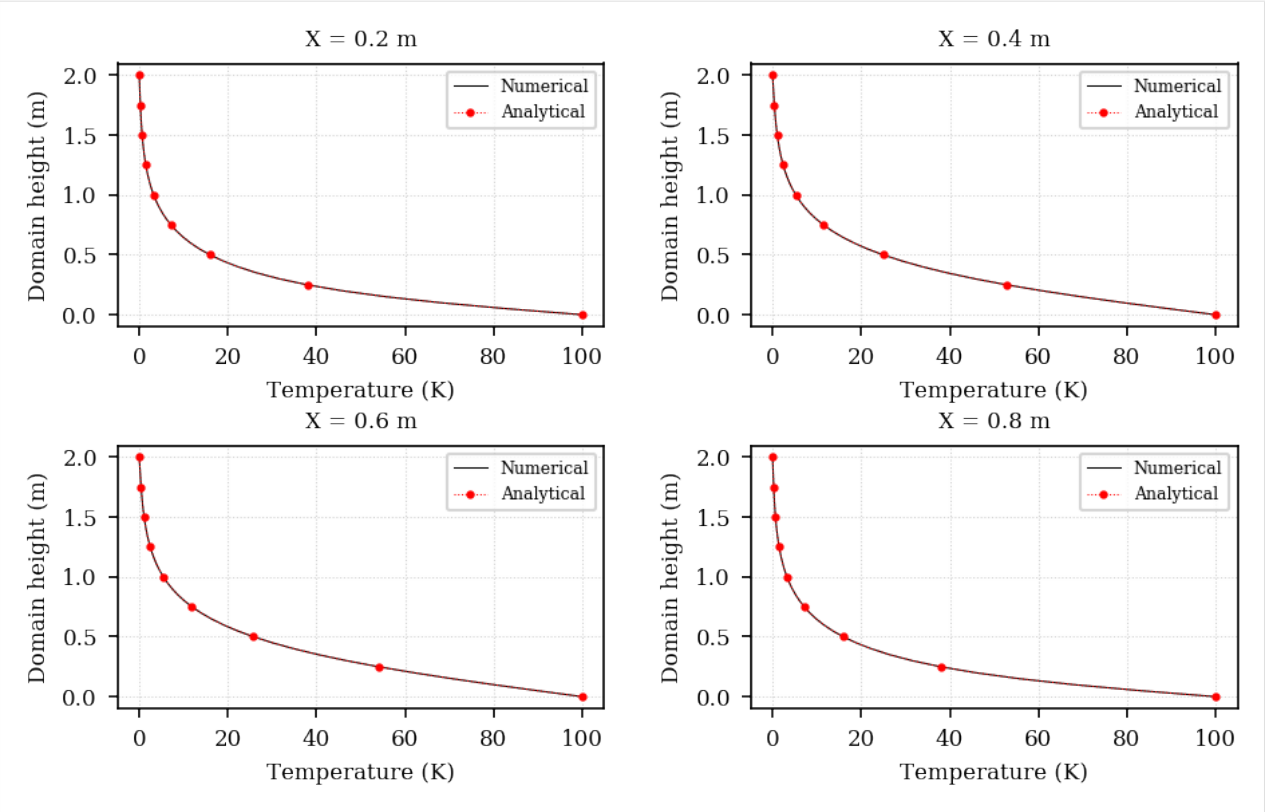

```
Numpy package test SUCCESS.
Matplotlib package test SUCCESS.
Math package test SUCCESS.
```

2.3 Test # 3

Lestly, we need to verify that the important components of nanpack are working. Enter the command shown in cell 3 (replace with correct project path) and verify that the test is completed successfully as shown in the ouput below.

[3]:
```
%run -m nanpack.tests.test_run
```

```
Uniform rectangular grid generation in cartesian coordinate system: Completed.
2
Assigning COLD-START initial conditions to the dependent term.
Initialization: Completed.
   ITER           ERROR
   ----           -----
      2    340.19325284
      4     41.00275596
      6     14.14011094
      8      6.69943966
     10      3.00219428
     12      1.33465332
     14      0.60100640
     16      0.27475525
     18      0.12717846
     20      0.05943372
     22      0.02797837
     24      0.01324526
Starting calculations to obtain analytical solution.
Calculating analytical solution: Completed.
Test run execution SUCCESS.
```



2.4 What to do next?

After these tests are successful, you may proceed to running some example scripts that you may create yourself using the tutorials included in the documentation.

If the tests are not successful, here are some options to look into:

1. If Test # 1 failed, try re-installing package. Make sure that the Python is included in your environment path variable. Please bring it to the attention of the author by raising an issue on GitHub or by messaging me on Twitter (@_NAnPack) or on LinkedIn NAnPack Community group.
2. If Test # 2 failed, check which package is not working and re-install the package.
3. If Test # 1 and Test # 2 passed, then Test # 3 must complete successfully too. If it doesn't please bring the issue to the attention of the author by raising an issue on GitHub or by messaging me on Twitter (@_NAnPack) or on LinkedIn NAnPack Community group.

CHAPTER

THREE

USAGE

The NAnPack Learners edition is intended to be used for the purpose of teaching and learning various numerical schemes for model differential equations in engineering and science. Presently, the capability of the package is limited to solving fluid and heat transfer equations such as diffusion equation, Poisson's equation, wave equation, inviscid and viscous Burgers equation.

Note:

For students:*Treat this as a supplementary learning material to learn simulating physics of the flow.*

For instructors:*Instructors may include this package in their teaching and can develop modules on top of this package based on their coursework requirements.*

3.1 Objective

The primary objective of NAnPack-Learners package is to impart learning in numerical schemes and coding with a simplified approach to those who want to develop a necessary background for Computational Engineering and Sciences, especially, in Computational Fluid Dynamics. It is expected of users that they will use the libraries in this package and write their own codes or scripts to run simulations and analyse the output. The only pre-requisite is to have an interest and some experience in coding, physics and calculus.

An important motivation behind the development of this package is to equip early career CFD enthusiasts with the complete know-how of the subject. Therefore, our aim is to provide complete information regarding what is going on behind the scenes when you run a simulation starting from the very basic.

3.2 Using NAnPack-Learners Package

Use NAnPack-Learners package to develop a fundamental background in computational methods in engineering. The package is designed in such a manner that the users are provided with the necessary tools and they are expected to build and execute their scripts using the package modules to perform numerical experimentations. Background on numerical schemes, physics, and built-in functions are presently provided in the tutorials section below.

Users may follow the instructions and examples to learn to develop their codes and do investigations of their own. Users may also submit their examples through GitHub pull requests and contribute to the documentation. The developers of this package are looking forward to seeing your fun projects that you will do using NAnPack.

Please note that in future the tutorials will be moved to the blog section. We recommend users to follow/comment or start a discussion in the repository page to stay updated.

Follow the tutorials in the next section to develop a deeper understanding of this package.

CHAPTER

FOUR

TUTORIALS

4.1 Tutorial 1. Understanding Configuration File

[1]:
```
# Document Author: Dr. Vishal Sharma
# Author email: sharma_vishal14@hotmail.com
# License: MIT
# This tutorial is applicable for NAnPack version 1.0.0-alpha4
```

For installation instructions, see Installation page.

The very first step to start using this package is to get familiar with the configuration file "config.ini". It is expected that the package users are familiar with the general terms, model equations etc. used in engineering simulations.

The configuration file is used as a tool to set-up the scenario that is to be solved numerically. The file accepts several inputs that are required in the pre-processing and post-processing steps of the simulation. Although users may choose to define the inputs in the scripts that they will develop without having to use the config file, however, it is highly recommended to set-up the numerical experiments using this configuration file. By doing so, users, particularly the starters in CFD, will get an idea about what information they will require beforehand in their experimentation as well as they will be able to make the best use of this package.

The structure of the config file includes the section name in square paranthesis [SECTION-NAME] and each section consists of key-value pairs in the format KEY = VALUE. The keys must not be changed by the user unless specified and user defined inputs must be specified in the value fields.

4.1.1 Section - [SETUP]

Provide the various experiment related description as inputs in this section.

```
[SETUP]

    EXPID        =   xxxxxxxx-xx
    UNITS_SYSTEM=   BRITISH or SI
    DESCRIPTION =   enter short description
    STATE        =   TRANSIENT or STEADY-STATE
    MODEL        =   FO_WAVE
    SCHEME       =   RUNGE-KUTTA
    DIMENSION    =   1D
```

'EXPID': Experiment ID- a unique ID that you can assign to your experiments.

I always prefer to include an ID for my records so that I can distinguish my experiment outputs from one another and for any changes I make in experiment set-up, I increment this ID number. Typically, an ID may be of the format

MMDDYYYY-Serial No. There may be several usages of this id, such as users may make a folder using this ID and save the output with the associated configuration file in this folder and repeat this process for each experiment. Adopting a Unique ID assigning strategy in the experiments help in efficient record management which is very important when publishing data or for reproducibility.

'UNITS_SYSTEM': Mention the system of units used in the simulation. This field helps in keeping track of the unit and to stay consistent with the system. It is also wise to save the unit system for the records.

'DESCRIPTION': Enter a short case description such as STEADY STATE HEAT CONDUCTION SIMULATION. Datatype = *string*.

'STATE': Allowed inputs are STEADY-STATE or TRANSIENT. The field represents the state at which the results are desired. Depending on the case, a value must be entered. This is a required field to calculate several simulation parameters by the program setup.

'MODEL': Classification of model equation. Allowed inputs are DIFFUSION, WAVE, FO_WAVE, VISC_BURGERS, INV_BURGERS. This field is required by the program setup. Please note that these model equations require only one dependent variable to be solved along X or along X and Y at each iteration level.

'SCHEME': User may choose to specify the numerical method they will be using in the simulation. This field is optional in the current version.

'DIMENSION': Allowed input 1D or 2D, depending on the simulation domain. This is required by the program setup.

4.1.2 Section - [DOMAIN]

The domain specification is entered in this section depending on 1D or 2D simulation.

```
[DOMAIN]

    LENGTH      =   400.0
    HEIGHT      =   0.0
```

'LENGTH': Length of the domain (along X axis). Value required. Datatype = *float*.

'HEIGHT': Height of the domain (along Y axis). If DIMENSION = 2D, value required. Datatype = *float*.

4.1.3 Section - [MESH]

Provide the details for meshing in this section.

```
[MESH]

    GRID_FROM_FILE? =   NO
    GRID_FNAME      =   none
    GRID_AUTO_CALC? =   YES
    dX              =   5.0
    dY              =   0.05
    iMax            =   0
    jMax            =   0
```

'GRID_FROM_FILE?': Read grid data from input file? This version does not support grid input through file, therefore keep the value as NO.

'GRID_FNAME': Grid input file name. This field is used to enter the file name for the grid input. Since, we have specified GRID_FROM_FILE as NO, leave it as 'none'.

'GRID_AUTO_CALC?': Auto-calculate grid points? Enter YES or NO. If mentioned YES, grid step size- (dX) or (dX, dY) must be entered and the program will auto-compute the grid points in the mesh. If mentioned NO, maximum grid points in the mesh- (iMax) or (iMax, jMAx) must be specified. For beginners, enter YES and specify dX, dY values.

Datatype for dX, dY = *float*.
Datatype for iMax, jMax = *integer*.

Please note that the current version supports only uniform, finite difference grid in the simulation which is good for applications with rectangular, simply connected domain. Advanced techniques will be introduced in subsequent versions.

4.1.4 Section - [IC]

Provide the option for the starting point (initial conditions) of the simulation.

```
[IC]

    START_OPT       =   COLD-START
    RESTART_FILE    =   none
```

'START_OPT': Starting option. Allowed inputs are COLD-START or RESTART. In the present version, only COLD-START feature is available which allows the user to start the simulation from zero initial values or using user developed subroutine for initial conditions.

RESTART conditions are useful when it is desired to start the simulation from the previously stored solution, for example, consider a scenario where your simulation ran for 24 hours or thousands of iterations without converging and your application crashed or reached a maximum limit of iterations, will it be efficient to run the simulation again from the beginning or by using previously stored solution as the starting point? Another scenario- starting a simulation from a converged solution to test something new will help in faster convergence.

'RESTART_FILE': Restart file name. File name for the program to read the stored solution. If START_OPT = COLD_START, leave it as none, otherwise specify the file name.

4.1.5 Section - [BC]

Provide the information whether to read the boundary conditions from a configuration file.

```
[BC]

    BC_FROM_FILE?   =   NO
    BC_FILE_NAME    =   none
```

'BC_FROM_FILE?': Read boundary conditions from file? Allowed inputs are YES or NO. First time users- enter NO.

There is a "bc.ini" file included in this package download, however, it is recommended to write a function to assign boundary conditions. There will be a separate tutorial on the boundary condition specification through "bc.ini" file.

'BC_FILE_NAME': Boundary condition input file name. If BC_FROM_FILE? = YES, the program will read the stored boundary conditions from file. If BC_FROM_FILE = NO, leave it as none.

4.1.6 Section - [CONST]

Provide the information to specify the constants in the model equations.

```
[CONST]

    CFL             =   1.0
    CONV            =   250.0
    DIFF            =   0.0
```

The program uses these constants to calculate coefficients in the finite difference approximations.

'CFL': In nanpack we use the term CFL to represents the constant coefficient in the finite difference formulation. (This is not a true definition of CFL though). For a diffusion model, the program will treat the CFL as diffusion number to obtain time step size and in a wave equation or convection models, in general, the program will treat the CFL as the Courant number. The CFL must satisfy the corresponding stability requirement, otherwise, the solution will not converge or will fail when late time solutions are required. This is a required field. Datatype = *float*.

'CONV': Convection coefficient. This is a required field for convection models such as WAVE equation. Datatype = *float*.

'DIFF': Diffusion coefficient. This is a required field for diffusion models such as DIFFUSION equation. Datatype = *float*.

4.1.7 Section - [STOP]

Provide the information about stopping simulation.

```
[STOP]

    SIM_TIME        =   0.45
    CONV_CRIT       =   0.01
    nMAX            =   3000
```

*It is always helpful to restrict the simulation time or iterations to terminate the program without crashing it. Consider a scenario when the solution has converged but it continues to solve the equations because the user did not set a break point and thus, the simulation has to be stopped somehow. Consider another scenario when you desire time-dependent solution but you have to do hand computations to calculate the required time-steps. Such scenarios can be avoided by specifying values in this section and let the program handle the termination process.

'SIM_TIME': Simulation time, required field for time-dependent solution. Datatype = *float*.

'CONV_CRIT': Convergence criteria, required field for steady-state solution. Datatype = *float*.

'nMax': Maximum iterations/time levels to terminate the program if solution didn't converge. This is a required field. Datatype = *integer*.

4.1.8 Section - [OUTPUT]

Provide information about saving output or monitoring convergence.

```
[OUTPUT]

    HIST_FILE_NAME  =   ./output/HISTfct1D.dat
    RESTART_FNAME   =   none
    RESULT_FNAME    =   ./output/fct1D.dat
    WRITE_EVERY     =   10
    DISPLAY_EVERY   =   10
    SAVE_FOR_ANIM?  =   NO
    SAVE_EVERY      =   10
    SAVE_1D_OUTPUT? =   YES
    X               =   0.2,0.4,0.6,0.8,1.0
    SAVE1D_FILENAME =   ./output/fo-up1Dx.dat
```

'HIST_FILE_NAME': Convergence history file name. The convergence history will be stored in this file. This is required if the user wants to store convergence data.

'RESTART_FNAME': File to create a restart point. This file may be used later when the user wants to restart the solution from the stored solution. This feature is not supported in the present version.

'RESULT_FNAME': Output file name. The solution of the dependent variable will be stored in this file at each grid point locations within the domain. This is a required field.

'WRITE_EVERY': Write solution file after every how many iterations? Datatype = *integer*.

For example, value = 10 means that the solution will be saved to files after every 10 iteration steps. This is a required field. To optimize the computational processing, use larger values depending on the problem.

'DISPLAY_EVERY': Write and display convergence history after every how many iterations? Datatype = *integer*.

'SAVE_FOR_ANIM?': Save intermediate solutions in separate files for animation? Allowed inputs are YES or NO. This feature is not available in the current version, thus enter NO.

'SAVE_EVERY': Save files for animation after every how many iterations? Datatype = *integer*. This field is required if SAVE_FOR_ANIM = YES.

'SAVE_1D_OUTPUT': Save output in 1D format along X or Y in 2D simulation? Allowed inputs are YES or NO.

While validating numerical methods, it is important to plot line graphs to compare the output with the known analytical solution to benchmark the method. In such cases, plotting the colorful contour plots does not help and thus solution of the dependent variable along an axis is required, for example u(x=0.2, y) can be plotted to perform a detailed analysis. It is recommended to use this feature for 2D simulations.

'X' or ***'Y'***: This key must be changed based on direction of the nodes at which the 1D solution is desired to be saved. The values to the key are the X or Y locations. Datatype = *float*.

Example: If the user wants to save the solution for u at x = 0.2, 0.4, 0.6 locations such that u(x=0.2, y), u(x=0.4, y), u(x=0.6, y), type the key as X and the values as 0.2, 0.4, 0.6 (separated by ','). Vice-versa, to save u(x, y=0.3), u(x,y=0.6), u(x, y=0.9), type the key as 'Y' and the values as 0.3, 0.6, 0.9.

'SAVE1D_FILENAME': 1D output file name. The solution of the dependent variable along the specified axis and locations will be stored in this file. This is required if SAVE_1D_OUTPUT = YES.

4.2 Tutorial 2. Solving a 1D diffusion equation

[ ]:
```
# Document Author: Dr. Vishal Sharma
# Author email: sharma_vishal14@hotmail.com
# License: MIT
# This tutorial is applicable for NAnPack version 1.0.0-alpha4
```

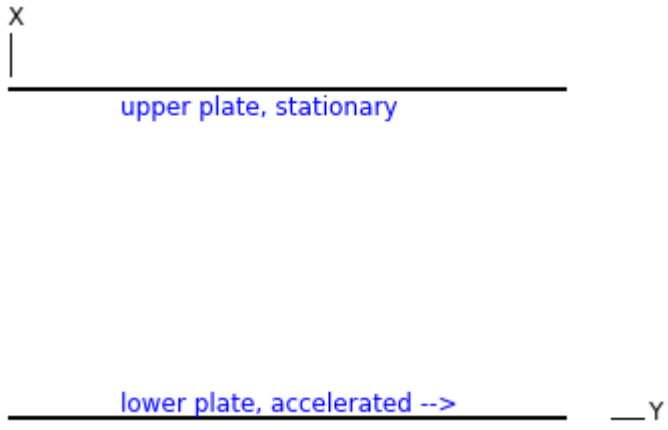
4.2.1 I. Background

The objective of this tutorial is to present the step-by-step solution of a 1D diffusion equation using NAnPack such that users can follow the instructions to learn using this package. The numerical solution is obtained using the Forward Time Central Spacing (FTCS) method. The detailed description of the FTCS method is presented in Section IV of this tutorial.

4.2.2 II. Case Description

We will be solving a classical probkem of a suddenly accelerated plate in fluid mechanicas which has the known exact solution. In this problem, the fluid is bounded between two parallel plates. The upper plate remains stationary and the lower plate is suddenly accelerated in y-direction at velocity U_o. It is required to find the velocity profile between the plates for the given initial and boundary conditions.

(For the sake of simplicity in setting up numerical variables, let's assume that the x-axis is pointed in the upward direction and y-axis is pointed along the horizontal direction as shown in the schematic below:



Initial conditions

$$u(t = 0.0, 0.0 < x \leq H) = 0.0\ m/s$$

$$u(t = 0.0, x = 0.0) = 40.0\ m/s$$

Boundary conditions

$$u(t \geq 0.0, x = 0.0) = 40.0\ m/s$$

$$u(t \geq 0.0, x = H) = 0.0\ m/s$$

Viscosity of fluid, $\nu = 2.17 * 10^{-4}\ m^2/s$
Distance between plates, $H = 0.04\ m$
Grid step size, $dx = 0.001\ m$
Simulation time, $T = 1.08\ sec$

Specify the required simulation inputs based on our setup in the configuration file provided with this package. You may choose to save the configuration file with any other filename. I have saved the configuration file in the "input" folder of my project directory such that the relative path is `./input/config.ini`.

4.2.3 III. Governing Equation

The governing equation for the given application is the simplified for the the Navies-Stokes equation which is given as:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2}$$

This is the diffusion equation model and is classified as the parabolic PDE.

4.2.4 IV. FTCS method

The forward time central spacing approximation equation in 1D is presented here. This is a time explicit method which means that one unknown is calculated using the known neighbouring values from the previous time step. Here *i* represents grid point location, *n*+1 is the future time step, and *n* is the current time step.

$$u_i^{n+1} = u_i^n + \frac{\nu \Delta t}{(\Delta x)^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

The order of this approximation is $[(\Delta t), (\Delta x)^2]$

The diffusion number is given as $d_x = \nu \frac{\Delta t}{(\Delta x)^2}$ and for one-dimensional applications the stability criteria is $d_x \leq \frac{1}{2}$

The solution presented here is obtained using a diffusion number = 0.5 (CFL = 0.5 in configuration file). Time step size will be computed using the expression of diffusion number. Beginners are encouraged to try diffusion numbers greater than 0.5 as an exercise after running this script.

Users are encouraged to read my blogs on numerical methods - link here.

4.2.5 V. Script Development

Please note that this code script is provided in file ``./examples/tutorial-02-diffusion-1D-solvers-FTCS.py``.

As per the Python established coding guidelines PEP 8, all package imports must be done at the top part of the script in the following sequence – 1. import standard library 2. import third party modules 3. import local application/library specific

Accordingly, in our code we will importing the following required modules (in alphabetical order). If you are using Jupyter notebook, hit `Shift + Enter` on each cell after typing the code.

[2]:
```
import matplotlib.pyplot as plt
from nanpack.benchmark import ParallelPlateFlow
import nanpack.preprocess as pre
from nanpack.grid import RectangularGrid
from nanpack.parabolicsolvers import FTCS
import nanpack.postprocess as post
```

As the first step in simulation, we have to tell our script to read the inputs and assign those inputs to the variables/objects that we will use in our entire code. For this purpose, there is a class `RunConfig` in `nanpack.preprocess` module. We will call this class and assign an object (instance) to it so that we can use its member variables. The `RunConfig` class is written in such a manner that its methods get executed as soon as it's instance is created. The users must provide the configuration file path as a parameter to `RunConfig` class.

[3]:
```
FileName = "path/to/project/input/config.ini" # specify the correct file path
cfg = pre.RunConfig(FileName) # cfg is an instance of RunConfig class which can be␣
↪used to access class variables. You may choose any variable in place of cfg.
```

```
************************************************************
************************************************************
Starting configuration.

Searching for simulation configuration file in path:
"D:/MyProjects/projectroot/nanpack/input/config.ini"
SUCCESS: Configuration file parsing.
Checking whether all sections are included in config file.
Checking section SETUP: Completed.
Checking section DOMAIN: Completed.
Checking section MESH: Completed.
Checking section IC: Completed.
Checking section BC: Completed.
Checking section CONST: Completed.
Checking section STOP: Completed.
Checking section OUTPUT: Completed.
Checking numerical setup.
User inputs in SETUP section check: Completed.
Accessing domain geometry configuration: Completed
Accessing meshing configuration: Completed.
Calculating grid size: Completed.
Assigning COLD-START initial conditions to the dependent term.
Initialization: Completed.
Accessing boundary condition settings: Completed
Accessing constant data: Completed.
Calculating time step size for the simulation: Completed.
Calculating maximum iterations/steps for the simulation: Completed.
Accessing simulation stop settings: Completed.
Accessing settings for storing outputs: Completed.

************************************************************
CASE DESCRIPTION                SUDDENLY ACC. PLATE
SOLVER STATE                    TRANSIENT
MODEL EQUATION                  DIFFUSION
DOMAIN DIMENSION                1D
    LENGTH                      0.04
GRID STEP SIZE
    dX                          0.001
TIME STEP                       0.002
GRID POINTS
    along X                     41
DIFFUSION CONST.                2.1700e-04
DIFFUSION NUMBER                0.5
TOTAL SIMULATION TIME           1.08
NUMBER OF TIME STEPS            468
START CONDITION                 COLD-START
************************************************************
SUCEESS: Configuration completed.
```

(continues on next page)

(continued from previous page)

You will obtain several configuration messages on your output screen so that you can verify that your inputs are correct and that the configuration is successfully completed. Next step is the assignment of initial conditions and the boundary conditions. For assigning boundary conditions, I have created a function `BC()` which we will be calling in the next cell. I have included this function at the bottom of this tutorial for your reference. It is to be noted that U is the dependent variable that was initialized when we executed the configuration, and thus we will be using `cfg.U` to access the initialized U. In a similar manner, all the inputs provided in the configuration file can be obtained by using configuration class object `cfg.` as the prefix to the variable names. Users are allowed to use any object of their choice.

If you are using Jupyter Notebook, the function BC must be executed before referencing to it, otherwise, you will get an error. Jump to the bottom of this notebook where you see code cell # 1 containing the ``BC()`` function

[4]:
```
# Assign initial conditions
cfg.U[0] = 40.0
cfg.U[1:] = 0.0

# Assign boundary conditions
U = BC(cfg.U)
```

Next, we will be calculating location of all grid points within the domain using the function `RectangularGrid()` and save values into X. We will also require to calculate diffusion number in X direction. In nanpack, the program treats the diffusion number = CFL for 1D applications that we entered in the configuration file, and therefore this step may be skipped, however, it is not the same in two-dimensional applications and therefore to stay consistent and to avoid confusion we will be using the function `DiffusionNumbers()` to compute the term `diffX`.

[5]:
```
X, _ = RectangularGrid(cfg.dX, cfg.iMax)
```

```
Uniform rectangular grid generation in cartesian coordinate system: Completed.
```

[6]:
```
diffX,_ = pre.DiffusionNumbers(cfg.Dimension, cfg.diff, cfg.dT, cfg.dX)
```

```
Calculating diffusion numbers: Completed.
```

Next, we will initialize some local variables before start the time stepping:

[7]:
```
Error = 1.0 # variable to keep track of error
n = 0 # variable to advance in time
```

Start time loop using while loop such that if one of the condition returns False, the time stepping will be stopped. For explanation of each line, see the comments. Please note the identation of the codes within the while loop. Take extra care with indentation as Python is very particular about it.

[8]:
```
while n <= cfg.nMax and Error > cfg.ConvCrit: # start loop
    Error = 0.0 # reset error to 0.0 at the beginning of each step
    n += 1 # advance the value of n at each step
    Uold = U.copy() # store solution at time level, n
    U = FTCS(Uold, diffX) # solve for U using FTCS method at time level n+1
    Error = post.AbsoluteError(U, Uold) # calculate errors
    U = BC(U) # Update BC
    post.MonitorConvergence(cfg, n, Error) # Use this function to monitor convergence
    post.WriteSolutionToFile(U, n, cfg.nWrite, cfg.nMax,\
                             cfg.OutFileName, cfg.dX) # Write output to file
    post.WriteConvHistToFile(cfg, n, Error) # Write convergence log to history file
```

```
   ITER           ERROR
   ----           -----
     10      4.92187500
     20      3.52394104
     30      2.88928896
     40      2.50741375
     50      2.24550338
     60      2.05156084
     70      1.90048503
     80      1.77844060
     90      1.67704721
    100      1.59085792
    110      1.51614304
    120      1.45025226
    130      1.39125374
    140      1.33771501
    150      1.28856146
    160      1.24298016
    170      1.20035213
    180      1.16020337
    190      1.12216882
    200      1.08596559
    210      1.05137298
    220      1.01821734
    230      0.98636083
    240      0.95569280
    250      0.92612336
    260      0.89757851
    270      0.86999638
    280      0.84332454
    290      0.81751777
    300      0.79253655
    310      0.76834575
    320      0.74491380
    330      0.72221190
    340      0.70021355
    350      0.67889409
    360      0.65823042
    370      0.63820074
    380      0.61878436
    390      0.59996158
    400      0.58171354
    410      0.56402217
    420      0.54687008
    430      0.53024053
    440      0.51411737
    450      0.49848501
    460      0.48332837

STATUS: SOLUTION OBTAINED AT
TIME LEVEL= 1.08 s.
TIME STEPS= 468

Writing convergence log file: Completed.
Files saved:
"D:/MyProjects/projectroot/nanpack/output/HISTftcs1D.dat".
```

In the above convergence monitor, it is worth noting that the solution error is gradually moving towards zero which

is what we need to confirm stability in the solution. If the solution becomes unstable, the errors will rise, probably upto the point where your code will crash. As you know that the solution obtained is a time-dependent solution and therefore, we didn't allow the code to run until the convergence is observed. If a steady-state solution is desired, change the STATE key in the configuration file equals to "STEADY" and specify a much larger value of nMax key, say nMax = 5000. This is left as an exercise for the users to obtain a stead-state solution. Also, try running the solution with the larger grid step size, Δx or a larger time step size, Δt.

After the time stepping is completed, save the final results to the output files.

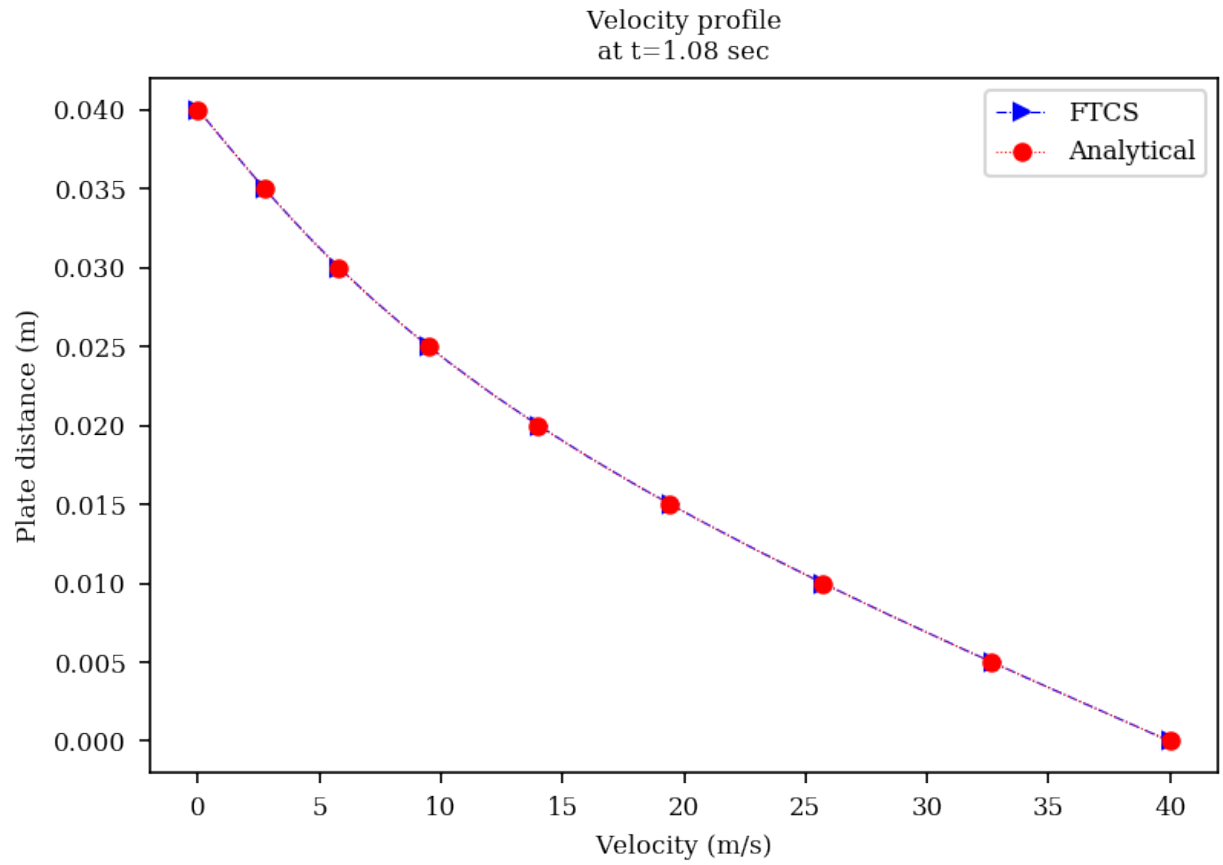
[9]:
```
# Write output to file
post.WriteSolutionToFile(U, n, cfg.nWrite, cfg.nMax,
                         cfg.OutFileName, cfg.dX)
# Write convergence history log to a file
post.WriteConvHistToFile(cfg, n, Error)
```

Verify that the files are saved in the target directory. Now let us obtain analytical solution of this flow that will help us in validating our codes.

[10]:
```
# Obtain analytical solution
Uana = ParallelPlateFlow(40.0, X, cfg.diff, cfg.totTime, 20)
```

Next, we will validate our results by plotting the results using the matplotlib package that we have imported above. Type the following lines of codes:

[11]:
```
plt.rc("font", family="serif", size=8) # Assign fonts in the plot
fig, ax = plt.subplots(dpi=150) # Create axis for plotting
plt.plot(U, X, ">-.b", linewidth=0.5, label="FTCS",\
         markersize=5, markevery=5) # Plot data with required labels and markers,␣
↪customize the plot however you may like
plt.plot(Uana, X, "o:r", linewidth=0.5, label="Analytical",\
             markersize=5, markevery=5) # Plot analytical solution on the same plot
plt.xlabel('Velocity (m/s)') # X-axis labelling
plt.ylabel('Plate distance (m)') # Y-axis labelling
plt.title(f"Velocity profile\nat t={cfg.totTime} sec", fontsize=8) # Plot title
plt.legend()
plt.show() # Show plot- this command is very important
```



Function for the boundary conditions.

[1]:
```
def BC(U):
    """Return the dependent variable with the updated values at the boundaries."""
    U[0] = 40.0
    U[-1] = 0.0

    return U
```

Congratulations, you have completed the first coding tutoria using nanpack package and verified that your codes produced correct results. If you solve some other similar diffusion-1D model example, share it with the nanpack community. I will be excited to see your projects.

[ ]:
```
# Document Author: Dr. Vishal Sharma
# Author email: sharma_vishal14@hotmail.com
# License: MIT
# This tutorial is applicable for NAnPack version 1.0.0-alpha4
```

4.3 Tutorial 3: Solving a 1D diffusion equation using all methods

4.3.1 I. Objectives

The objectives of this tutorial are two-fold: Firstly, inform users about the various available numerical methods for solving 1D diffusion equation and comparing the numerical solutions obtained from those methods, and Secondly, creating an automation script- that can run simulations using all available numerical method for 1D diffusion model so as to reduce user efforts that will go into writing multiple scripts. This automation example can be used as a reference for creating other automation scripts.

4.3.2 II. Case Description

We will use the same example which was presented in *Tutorial 2*.

4.3.3 III. Numerical Methods

1. **Forward Time Central Spacing (FTCS) method**

Brief description of this method is given in *Tutorial 2*.

Function call:

```
nanpack.parabolicsolvers.FTCS(Uo, diffX, diffY)
```

2. **DuFort-Frankel method**

This is an explicit method in which the time and space derivatives are discretized by second-order central differencing which is the same as in Richardson method, however, to make the scheme stable, the term u_i^n in the diffusion term is approximated by averaging over two time steps such that

$$u_i^n = \frac{u_i^{n+1} + u_i^{n-1}}{2}$$

Such modification makes the scheme unconditionally stable. After some modifications, the discretized equation is expressed as

$$[1 + 2d_x]u_i^{n+1} = [1 - 2d_x]u_i^{n-1} + 2d_x[u_{i+1}^n + u_{i-1}^n]$$

where

$$d_x = \frac{\nu(\Delta t)}{(\Delta x)^2}$$

As observed in the equation, values of the dependent variable at two time steps *n* and (*n*-1) is required, hence the storage requirements are increased. Also, the accuracy of the DuFort Frankel depends on the starter solution which depends on two sets of initial conditions. This method is second-order accurate in both space and time.

Function call:

```
nanpack.parabolicsolvers.DuFortFrankel(Uo, Uold2, diffX, diffY)
```

3. **Laasonen method**

This is an implicit formulation which is expressed as

$$Au_{i+1}^{n+1} + Bu_i^{n+1} + Cu_{i-1}^{n+1} = D$$

where,

$$A = -d_x$$

$$B = 1 + 2d_x$$

$$C = -d_x$$

$$D = u_i^n$$

$$d_x = \frac{\nu(\Delta t)}{(\Delta x)^2}$$

The implicit schemes are unconditionally stable and therefore a larger time step can be used to minimize the simulation steps. The time step is, however, restricted due to other numerical errors such as truncation error.

The discretized equation results in the set of linear algebraic equations. Subsequently, the algebraic equations are written in the matrix form that consists of a tridiagonal coefficient matrix. This formulation leads to larger computational time which can be somewhat compensated by using a larger time step.

Function call:

```
nanpack.parabolicsolvers.Laasonen(Uo, diffX)
```

4. **Crank-Nicolson method**

The Crank-Nicolson is also an implicit formulation in which the diffusion term is approximated by averaging the central difference at time levels *n* and *n*+1. The discretized equation is expressed as:

$$Au_{i+1}^{n+1} + Bu_i^{n+1} + Cu_{i-1}^{n+1} = D$$

where,

$$A = -\frac{1}{2}d_x$$

$$B = 1 + d_x$$

$$C = -\frac{1}{2}d_x$$

$$D = \frac{1}{2}d_x u_{i+1}^n + (1 - d_x)u_i^n + \frac{1}{2}d_x u_{i-1}^n$$

$$d_x = \frac{\nu(\Delta t)}{(\Delta x)^2}$$

The Crank-Nicolson method is second-order accurate in both space and time.

Function call:

```
nanpack.parabolicsolvers.CrankNicolson(Uo, diffX)
```

Important: It is to be noted that both Laasonen and Crank-Nicolson methods are inefficient for 2D applications because the coefficient matrix is pentadiagonal, the solution of which is very time-consuming.

Additional resources

1. Link to my blogs.
2. Computational Fluid Dynamics, Vol. 1 by Dr. Klaus Hoffmann- This book is very clear and informative.

4.3.4 IV. Script Development

This code script is provided in file `./examples/tutorial-03-diffusion-1D-solvers-all.py`

Most of the script remains the same as in *Tutorial 2* and therefore explanation is only provided on how you can automate to run all numerical methods in a single program without the need to write the same code multiple times.

[2]:
```
# Import modules
import nanpack.preprocess as pre
from nanpack.grid import RectangularGrid
import nanpack.parabolicsolvers as pb
import nanpack.postprocess as post
from nanpack.benchmark import ParallelPlateFlow

cfg = pre.RunConfig("path/to/project/input/config.ini")

X, _ = RectangularGrid(cfg.dX, cfg.iMax)
diffX,_ = pre.DiffusionNumbers(cfg.Dimension, cfg.diff, cfg.dT, cfg.dX)
```

```
**********************************************************
**********************************************************
Starting configuration.

Searching for simulation configuration file in path:
"D:/MyProjects/projectroot/nanpack/input/config.ini"
SUCCESS: Configuration file parsing.
Checking whether all sections are included in config file.
Checking section SETUP: Completed.
Checking section DOMAIN: Completed.
Checking section MESH: Completed.
Checking section IC: Completed.
Checking section BC: Completed.
Checking section CONST: Completed.
Checking section STOP: Completed.
Checking section OUTPUT: Completed.
Checking numerical setup.
User inputs in SETUP section check: Completed.
Accessing domain geometry configuration: Completed
Accessing meshing configuration: Completed.
Calculating grid size: Completed.
Assigning COLD-START initial conditions to the dependent term.
Initialization: Completed.
Accessing boundary condition settings: Completed
Accessing constant data: Completed.
Calculating time step size for the simulation: Completed.
Calculating maximum iterations/steps for the simulation: Completed.
Accessing simulation stop settings: Completed.
Accessing settings for storing outputs: Completed.

**********************************************************
CASE DESCRIPTION                SUDDENLY ACC. PLATE
SOLVER STATE                    TRANSIENT
MODEL EQUATION                  DIFFUSION
DOMAIN DIMENSION                1D
    LENGTH                      0.04
GRID STEP SIZE
    dX                          0.001
TIME STEP                       0.002
GRID POINTS
```

(continues on next page)

(continued from previous page)

```
    along X                     41
DIFFUSION CONST.                2.1700e-04
DIFFUSION NUMBER                0.5
TOTAL SIMULATION TIME           1.08
NUMBER OF TIME STEPS            468
START CONDITION                 COLD-START
*********************************************************
SUCEESS: Configuration completed.

Uniform rectangular grid generation in cartesian coordinate system: Completed.
Calculating diffusion numbers: Completed.
```

Create a list `func` that contains the reference to the the numerical methods. Also, since in our configuration file we can provide only one file name as the input that will lead to overwriting of results in that one file, we have to provide 4 file names to the program to save the results from different numerical methods in their respective files. Let's create a list `files` as shown in code cell 3.

[3]:
```
func = [pb.FTCS, pb.DuFortFrankel, pb.Laasonen, pb.CrankNicolson]
files = ["FTCS", "DuFortFrankel", "Laasonen", "CrankNicolson"]
```

Write a `for` loop to iterate over the functions provided in the list `func`. In this way, one numerical solution is obtained using one method at the required time step and after completion, the next numerical method will be executed and so on, until all the numerical methods in `func` list have been executed.

Since DuFort-Frankel method requires two sets of initial solution, one of which will be obtained using the FTCS method as can be seen in the codes. The accurcy of the DuFort-Frankel method depends on this starter solution and often this starter solution is provided by the analytical solution, if available.

[4]:
```
# Start a loop for 4 solver functions
for f in range(len(func)):
    # Define initial conditions
    cfg.U[0] = 40.0
    cfg.U[1:] = 0.0
    # Define boundary conditions
    U = BC(cfg.U)
    # Start iterations
    Error = 1.0
    n = 0

    while n <= cfg.nMax and Error > cfg.ConvCrit:
        Error = 0.0
        n += 1
        # DuFort Frankel will be executed in this block
        if f == 1:
            if n == 1: # at first-time step, obtain starter solutions
                Uold = U.copy() # initial condition for n= -1 time level
                U = pb.FTCS(Uold, diffX) # FTCS solution for n=0 time level
            Uold2 = Uold.copy() # Store solution at (n-1)th time step
            Uold = U.copy() # Store solution at (n)th time step
            U = func[f](Uold, Uold2, diffX)
        # All other numerical methods will be executed in this block
        else:
            Uold = U.copy()
            U = func[f](Uold, diffX)
        Error = post.AbsoluteError(U, Uold)
        # Update BC
```

(continues on next page)

(continued from previous page)

```
        U = BC(U)
        # Write output to file
        # Provide a complete path where the files will be stored
        fname = f"path/to/project/output/{files[f]}1D.dat"
        convfname = f"path/to/project/output/hist{files[f]}1D.dat"
        post.WriteSolutionToFile(U, n, cfg.nWrite, cfg.nMax, fname, cfg.dX)
        # Write convergence history log to a file
        post.WriteConvHistToFile(cfg, n, Error, convfname)

    # Write output to file
    post.WriteSolutionToFile(U, n, cfg.nWrite, cfg.nMax, fname, cfg.dX)
    # Write convergence history log to a file
    post.WriteConvHistToFile(cfg, n, Error, convfname)
    print()
```

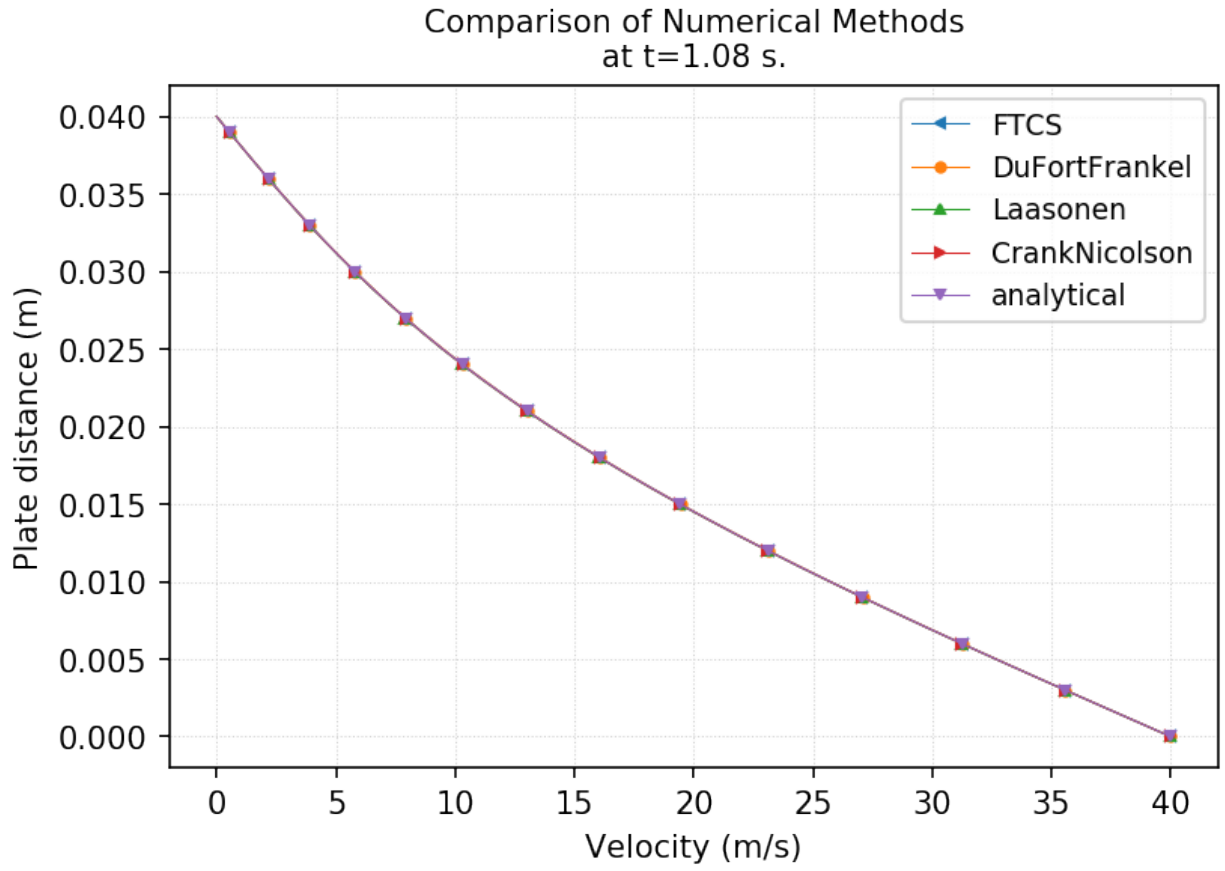
```
STATUS: SOLUTION OBTAINED AT
TIME LEVEL= 1.08 s.
TIME STEPS= 468

Writing convergence log file: Completed.
Files saved:
"D:/MyProjects/projectroot/nanpack/output/histFTCS1D.dat".


STATUS: SOLUTION OBTAINED AT
TIME LEVEL= 1.08 s.
TIME STEPS= 468

Writing convergence log file: Completed.
Files saved:
"D:/MyProjects/projectroot/nanpack/output/histDuFortFrankel1D.dat".


STATUS: SOLUTION OBTAINED AT
TIME LEVEL= 1.08 s.
TIME STEPS= 468

Writing convergence log file: Completed.
Files saved:
"D:/MyProjects/projectroot/nanpack/output/histLaasonen1D.dat".


STATUS: SOLUTION OBTAINED AT
TIME LEVEL= 1.08 s.
TIME STEPS= 468

Writing convergence log file: Completed.
Files saved:
"D:/MyProjects/projectroot/nanpack/output/histCrankNicolson1D.dat".
```

[5]:
```
# Obtain analytical solution
Uana = ParallelPlateFlow(40.0, X, cfg.diff, cfg.totTime, 20)
post.WriteSolutionToFile(Uana, 10, cfg.nWrite, cfg.nMax,
                         "path/to/project/output/analytical1D.dat",
                         cfg.dX)
```

Plot the results using the `Plot1DResults` function included in the package. Use `help(Plot1DResults)` command to see the allowed input arguments.

[6]:
```
fn = [] # empty list to store full path to the files
files.append("analytical") # add another file name to the files list
for f in range(len(files)): # add complete path to files from which the plotting␣
↪function will read data to plot
    fn.append(f"path/to/project/output/{files[f]}1D.dat")
# Call the plotting function and provide arguments to customize plot
post.Plot1DResults(dataFiles=fn, uAxis="X", Markers="default", Legend=files,\
                   Title=f"Comparison of Numerical Methods\nat t={cfg.totTime} s.",
                  xLabel="Velocity (m/s)", yLabel="Plate distance (m)")
```

```
Preparing data to plot results...
Plotting 1D results
```



[1]:
```
def BC(U):
    """Return the dependent variable with the updated values at the boundaries."""
    U[0] = 40.0
    U[-1] = 0.0

    return U
```

Congratulation, you have created a script to run all the available numerical solvers for 1D diffusion model and compared the numerical results using plotting tools.

CHAPTER

FIVE

CREDITS

I would like to acknowledge that most modules of this package including the numerical methods, their description and the solved examples, are created using using the information in the book written by my PhD advisor Dr. Klaus A. Hoffmann [1] and using my notes from AE-719 Computational Fluid Dynamics, AE-919 Advanced Computational Fluid Dynamics and AE-812 Viscous Fluid Flow courseworks at Wichita State University.

[1] KA Hoffmann and ST Chaing, Computational Fluid Dynamics, Vol.1 and Vol.2, 4th ed., 2004.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search