Practical notes on selected numerical methods with examples

Research Report Mech 312/15

Kert Tamm, Martin Lints, Dmitri Kartofelev, Päivo Simson,
Mart Ratas, Pearu Peterson
Institute of Cybernetics at Tallinn University of Technology

This collection of documents is aimed at approximately MSc level student. The goal is to give sufficient information in a reasonably compact format to be able to solve some types of partial differential equations numerically and to provide working example cases to help understand the methods from the practical standpoint. The present collection is not a substitute for a textbook dedicated to the relevant numerical method.

The structure of the collection is following:

(I) Practical application of pseudospectral method and visualizing the numerical solutions.
- Kert Tamm, 1D pseudospectral method implementation on Boussinesq like equation

(II) 2D Pseudospectral example
- Kert Tamm, Mart Ratas, Pearu Peterson, 2D Heat equation example

(III) Wavenumber filtering in pseudospectral methods
- Martin Lints, Accurcay and limits on pseudospectral method

(IV) A Practical Guide to Solving 1D Hyperbolic Problem with Finite Element Method
- Dmitri Kartofelev, Päivo Simson, Wave equation implementation with FEM.

TALLINN 2015

1 Practical application of pseudospectral method and visualizing the numerical solutions.

Kert Tamm, kert@ioc.ee

The purpose of the present document is to give approximately master level student who might not be very fluent in Python and Matlab necessary information in reasonably compact form for implementing pseudospectral method (PSM) and ability to visualise the results at level which might be acceptable for MSc or starting phase of PhD thesis. The style used is relatively personal and the document should be taken with grain of salt and some common sense as it contains few personal opinions.

The theoretical foundations of the method are described in [1] with acceptable detail. The document at hand focuses on practical applications of the described numerical method. The key points being that we use discrete Fourier transform and that implies that we are also using the discrete frequency function ω. Some data visualisation techniques are also described which can be applied independently of the numerical method used for solving the model equations. All examples and applications are in the 1D case in the present document. The examples are mix of Python and Matlab languages, the language used is noted at the location of the example. The key difference to keep in mind in the examples is that Matlab index starts with 1 while Python index starts at 0 for the vectors/arrays.

One of the assumptions we do in a nutshell is that locally and over short enough time intervals the process is "linear enough". Just something to keep in mind if your nonlinear parameters are starting to creep close to the rest of your equation parameters in magnitude. At the end of the day in physics the measure of reality is experiment so if it is possible at all check your results against known experimental results. Failing that check at least with normalized wave equation and make sure amplitude and speed of the wave is as expected.

Check also:

`http://digi.lib.ttu.ee/i/?582`
`www.ioc.ee/lints/MSc_Martin_Lints.pdf`
`www.ioc.ee/lints/PsM_precision_MLints2015.pdf`

These links are my PhD thesis (PSM application on Boussinesq like equation with mixed partial derivatives) the MSc thesis (in particular the sections about the accuracy and the application limits of the PSM) of Martin Lints and some notes on filtering by Martin Lints (included as third document in the present collection) who is currently a PhD student of prof. Andrus Salupere who was also my supervisor during my BSc, MSc and PhD studies.

PSM is a tool. Like with all tools user should be aware what can and can not be done with it and prerequisite for proper application is thinking it first through what is the goal and how to apply the tool properly. The worst way to apply a numerical method is the "black box" method meaning that user gives some input, pushes the button and something comes out which must be true because it came from computer. There is a special term for that kind of approach in engineering which is "Computer Assisted Catastrophe". Unless one writes ALL his code and librarys him/her-self some level of black box is unavoidable - in the present examples, for example, the ODE solver used is "Black Box" as well as Fast Fourier Transform implementations already existing in the used languages or packages. Fortunately these are reasonably well documented and going over that documentation is encouraged to be able to answer some inconvenient question about these which will be asked sooner or later. Do not apply the provided code samples like a black box - think through and understand them as well as you can.

1.1 The Pseudospectral method in a nutshell

The idea in a nutshell is relatively simple. You have to get your equation into a format where all time derivatives are on one side and the opposing side contains only space derivatives. Then you can make your partial differential equation (PDE) into ordinary differential equation (ODE) by using the Fourier transform. There is two possibilities: (1) solve it in the Fourier space and then do the inverse transform for getting back to the real space and (2) flip back and forth between Fourier space and real space each timestep. We use the second option (it works better for that kind of equation as there is no *fast* convulution algorithm to deal with nonlinearities without leaving the Fourier space).

ODE is usually a lot simpler to solve than PDE and you can use any of the existing ODE solvers.

Without getting into gritty details in practice one uses usually some-kind of already implemented FFT and IFFT functions (both Python and Matlab have these). If there arises any kind of problems using these it is good idea to read the documentation as different implementations can change in small but relevant details. As long as you are using the forward and inverse discrete Fourier transforms from the same package there should be no practical problems. If you are using different spatial length than 2π check if your FFT and IFFT functions are aware of that fact (usually an optional input parameter) as this is important. In Python:

```
import numpy
import scipy
from scipy.fftpack import *

fft_r = fft(temp2,n,-1) # forward transform
ru = ifft(fft_r).real # reverse transform

u0 = Ao/(cosh(Bo*xx))**2 # sech**2 pulse amplitude Ao, width Bo
u0x = diff(u0,period=2*numpy.pi*loik); #speed is -c*u0x
u0xx = diff(u0x,period=2*numpy.pi*loik);# second derivative by space
```

Few notes - diff (in scipy.fftpack) is Fourier transform based scheme for taking numerical derivatives by default. The variable "loik" is number of 2π section in your space interval, "n" is number of grid points. Just flat out forward and inverse Fourier transforms do not need that parameter in default implementation. Only real part is taken in the inverse transform for getting rid of practically zero imaginary part which can arise under some parameter combinations as a result of limited machine precision and accumulating errors (especially if you have high rank derivatives in your equations).

It is important to note that in Matlab the "diff" is implemented differently (not based on FFT) and reduces the length of your vector by one! The "fft" and "ifft" implementation is basically the same and in the simplest case takes just a vector as an argument. If the number of points in the vector is not power of 2 (2^n where n is number of grid-points) the algorithm takes significant performance penalty regardless of the language used.

Implementation example in Python. Normalized wave equation

```
import numpy
import scipy
from scipy.fftpack import diff
from scipy.integrate import ode

def cosh(x): # hyperbolic cosine
```

```
    sisene = numpy.exp(x)/2+1/(2*numpy.exp(x))
    return sisene

def InitialCondition(Ao,Bo,x,loik,c): # Sech**2 initial condition
    xx = x-loik*numpy.pi #profile shift
    u0 = Ao/(cosh(Bo*xx))**2 #initial pulse
    u0x = diff(u0,period=2*numpy.pi*loik); #speed -c*u0x
    yx[:n] = u0 # space
    yx[n:] = -c*u0x; # speed
    return yx.real

def EQ(t,yx): # equation to be solved
    nn = len(yx)/2
    ru = yx[:nn] # space displacement
    v = yx[nn:] # speed, or dr/dt, y[n] to y[2n-1]
    x_xx = diff(ru,2,period=2*numpy.pi*loik)  # second spatial derivative
    ytx[:nn] = v #du/dt=v
    ytx[nn:] = x_xx # wave equation utt = c* uxx, c=1 in normalized equation
    return ytx

n = 2**12 #number of grid-points
loik = 64 # number 2 pi sections in space
dt = 1 #time step
narv = 6001 # number of time-steps
x = numpy.arange(n,dtype=numpy.float64)*loik*2*numpy.pi/n #coordinate x
yx = numpy.arange(n*2,dtype=numpy.float64)
ytx = numpy.arange(n*2,dtype=numpy.float64)
c = 1 # speed squared of wave equation
Bo = 1/8; Ao = 1; # initial pulse param

yx = InitialCondition(Ao,Bo,x,loik,c);
t0 = 0.0; t_end = narv*dt #final time to integrate to
tvektorx = []; tulemusx = []; #arrays for results
runnerx = ode(EQ) #using ode from scipy
runnerx.set_integrator('vode',nsteps=1e7,rtol=1e-10,atol=1e-12);
runnerx.set_initial_value(yx,t0);
while runnerx.successful() and runnerx.t < t_end: #integration
      tvektorx.append(runnerx.t);
      tulemusx.append(runnerx.y);
      print('z ',runnerx.t);
      runnerx.integrate(runnerx.t+dt);
```

This *should* work. If it does not let me know. This is copy-pasted together from a much larger program so I might have forgotten to carry over something I'm using. This code example does not visualize and note that the resulting array contains **both** space and speed side by side with the time-step defined by the variable "dt".

1.1.1 Scaled wavenumber

Fourier transfrom, by default, assumes that the spatial length is 2π. If this is not the case then for a practical application it is advisable to use scaled wavenumber (rarely called also the

discrete frequency function) which is needed if you start taking derivatives "by hand" in Fourier space or if you have mixed partial derivatives in your governing equations and you need to do change of variables for making it possible to apply PSM for that kind of model equation. For a start few equations very shortly.

You have some-kind of equation you need to solve in a format:

$$A \cdot u_{tt} = B \cdot u_{xx} + C \cdot u_{xxtt} + D \cdot u_{xxxx}.$$

This is some-kind of Boussinesq-type equation, there is no nonlinearity in it but there are two dispersive terms including a mixed partial derivative term that prevents direct application of the PSM. Let's change it a bit for a start (it's not exactly needed but I like it that way).

$$u_{tt} = B/A \cdot u_{xx} + C/A \cdot u_{xxtt} + D/A \cdot u_{xxxx}.$$

Now lets move all terms containing time derivatives to the left hand side

$$u_{tt} - C/A \cdot u_{xxtt} = B/A \cdot u_{xx} + D/A \cdot u_{xxxx}.$$

Introducing new variable and using properties of Fourier transform

$$\Phi = u - C/A \cdot u_{xx};\ \Phi = \mathrm{F}^{-1}\left(\mathrm{F}(u)\right) - C/A \cdot \mathrm{F}^{-1}\left((i \cdot k)^2 \mathrm{F}\left(u\right)\right).$$

Then one can just solve the equation

$$\Phi_{tt} = B/A \cdot u_{xx} + D/A \cdot u_{xxxx},$$

returning to the "real space" by switching variables back whenever needed (if you do not you will get results which look like the results you expect but are, in fact, incorrect at the best case at least in amplitude). Meaning that you can express the original variable in terms the new function Φ as

$$u = \mathrm{F}^{-1}\left[\frac{\mathrm{F}(\Phi)}{1 + C/Ak^2}\right];\quad u_m = \mathrm{F}^{-1}\left[\frac{(i \cdot k)^m \mathrm{F}(\Phi)}{1 + C/Ak^2}\right].$$

Here subscript denotes partial differentiation rank and k is wave number (scaled wavenumber if the spatial length is not 2π). For just taking spatial derivatives in Fourier space $u_{x \cdot m} = \mathrm{F}^{-1}[(i \cdot k)^m \mathrm{F}(u)]$. For Φ note the different sign in denominator which results from the fact that $i^2 = -1$ flipping the sign. You could, in theory, somehow use the same logic for integration (by dividing with ik) but one should be very careful with that as in computer you get only one result and dividing with a complex number should, in reality have m valid values/roots so this is normally not done. In the Boussinesq type equations higher order even derivatives usually represent dispersive effects and odd derivatives dissipative effects. If you have more than 6th order derivatives in the model equation it is essential to keep eye on results in particularly critical way and if possible go for higher than double precision floating point (which usually means quite significant performance penalty) as it is probable that higher harmonics in your results have very little to do with reality because of limited machine precision.

Scaled wavenumber implementation in Python (in Matlab index is shifted by 1)

```
def oomega(n): #omega calculation
    qq = 0.0
    kk = 0
    while kk<(n/2):
        omega[kk] = qq/loik #first part
        omega[n/2+kk] = (-(n/2)+qq)/loik #second part
        qq=qq+1.0
        kk=kk+1
    return omega
```

Here "loik" is number of 2π sections in space and n is number of grid points. Spatial period is from 0 to loik * 2π, not from $-\pi$ to $+\pi$ as is assumed by default in Fourier transform.

1.1.2 Initial and boundary conditions

The boundary conditions **must** be periodical in the PSM based on Fourier transform. It is one of the limitations (or advantage, depending on how you look at it) of the method. There is other integral transforms that do not have this limitation like, for example, Laplace transform and it is possible to construct a spectral method based on these but they all have their drawbacks (for the Laplace transform it is complexity of the inverse transform if you have some-kind of not-nice function). The key advantage of the Fourier based PSM is the existence of **Fast** Fourier Transform algorithms, keyword being the "fast". It really is as long as number of grid-points n is 2^n and one has integer number of 2π spatial sections.

In implementation boundary conditions do not need to be defined or handled in any special way, they will be periodical because of using FFT based algorithm.

Do note that any discontinuities on the boundary are as inconvenient (if not even more, as they are easier to miss) as they are in the middle of ones space domain. For example the pulse type initial condition used mostly in this document in the form of hyperbolic secant does also have a small discontinuity at the boundary. Derivatives should be as continuous as possible as well, any no matter how small discontinuity is amplified each time a derivative is taken.

There is multiple possibilities of defining Fourier transform, the most common finite interval implementations are from $-\pi$ to π and from 0 to 2π. Depending on the implementation and if the spatial length is 2π or something else a correct discrete frequency function must be constructed. Here length 0 to $loik \cdot 2\pi$ is preferred. One of the reasons for that is to keep the spatial coordinates all positive.

Note about initial conditions is that some functions are defined around zero so if the spatial period is shifted to be starting from 0 it can be necessary to shift also the function (as is done with the sech-type initial pulse in the present examples).

You need one initial condition for each time derivative you have in the model equations. So for the regular wave equation (second order) two initial conditions are needed, usually displacement and speed. If you have u_{tttt} in your model you will need four initial conditions! Fortunately boundary conditions are periodic so you do not need to deal with these.

In practice it is possible to set all the higher order initial conditions to zero other than some kind of initial displacement (or whatever the interpretation of the lowest order initial condition). In essence this means that one is starting from the peak of interaction of two waveprofiles propagation in opposite directions (in the case of second order equation). The initial pulse, regardless of its shape, will split into two waveprofiles propagating in opposite directions with correct amplitude and speed, if its the linear model the amplitude must be exactly half of the initial pulse amplitude and speed must be exactly the speed of the sound in that environment if there is no dispersive terms, in the nonlinear models life can be much richer and 2+2 is not quite 4 most of the time. You can get away with the same approach for equations that are higher than 2nd order but it is not physically correct most of the time.

For initial condition in the examples an assumption has been made (which is usually done for evolution equations, like KdV (Korteweg de Vries) equation). The assumption is that one can move the solution into moving frame of reference so that $\xi = x - ct$, i.e, the solution "stands in place" as you ride along with it – this assumption is in essence incorrect for second order equation which contains nonlinearities as with second order equation you have two solutions propagating in opposite directions (you might get away with $\xi = x \pm ct$) and with nonlinearity the superposition principle does not hold anyway. But it works in practice so it is used which means that using these not entirely correct assumptions you can take the speed initial condition as $u_t = -c \cdot u_x$. It is trivial to find u_x and one gets some-kind of more or less reasonable initial condition.

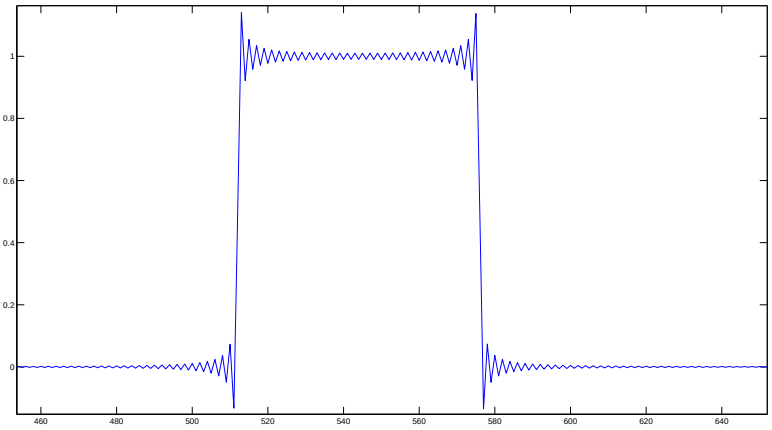
1.1.3 Filtering, accuracy and stability

For a start read the third section by Martin Lints. Filtering in Fourier space is something that is very common in various PSM implementations. One can get away without filtering as the examples in the present document are without any kind of filtering and they work. Reason why filtering is often used is that it makes the algorithm more stable and can be necessary especially if high number of grid-points or high rank derivatives are present. There is also some inherent dangers in filtering as with strong enough filter it is possible to stabilize a solution that **should not be stable**. In a sense that means with strong filter one can prevent, for example, shock-wave from forming under the conditions where it should form or prevent wave from breaking when it should break.

Computers have finite resolution. What is done, in essence, is taking a Fourier series which is in essence a infinite series and cutting it at some point saying that anything higher is already negligible enough to not count. If you have n grid-points the number of members (harmonics) in the discrete Fourier series is $n/2$. If you go over the texts pointed in the beginning by Martin Lints note that PSM is limited by both sides. At the lower end if you have too small number of grid points your accuracy suffers as you are trying to approximate whatever you are trying to simulate with small number of spectral components. On the other hand if you dial the number of grid-points too high you will run into trouble with machine precision with high harmonics, especially if you have high ranked derivatives present.

In practice the simplest form of filtering is dialling the highest harmonic to zero at each time-step. If this is not used it is advisable to keep an eye on highest harmonic at least and if it starts growing significantly take note that the result is no longer as accurate as it should. Reason for paying attention to the highest harmonic in your series is that it tends to collect the truncation error and as such can be the trouble-starter. Filtering breaks energy conservation as it takes energy away from the wave by dampening higher harmonics (good to keep in mind if you need to check for energy conservation). Normally the filter is some-kind of exponent suppressing some number of higher harmonics.

In general PSM tends to be computationally cheaper at the same accuracy than finite elements/differences based methods. It seems (that is an opinion, I have not checked it in rigorous way) that it also has smaller numerical dispersion than finite difference/elements based methods. There are *some* effects though, like for example Gibbs phenomenon one should be aware of (just Google it, English wiki article seems to be good enough at summarizing it).



This is a regular wave equation with rectangular function pulse propagating to the right demonstrating the Gibbs phenomenon.

What Gibbs phenomenon means in practice is that if your solution gets steep slopes it will start generating high frequency oscillations which are **not physical**, note that leading oscillations in the example solution travel faster than the speed of sound (or light) in the model. Gibbs phenomenon is not specific to PSM and exists in most numerical schemes.

1.1.4 Random notes and ramblings

Python is object oriented language. What this means in practice is that if you do

```
A = B
# some code happens here and suddenly you do:
B = C
# some more stuff happens here where you change the content of C
```

then A changes as a result of you changing the content of C! In practice this means that you can have some stuff "leaking" into places which you are not expecting if you are sloppy with variables.

Examples of inverse transform and solvable function in the case of mixed partial derivative in Python. Equation is

$$\frac{\partial^2 U}{\partial T^2} = (1 + PU + QU^2)\frac{\partial^2 U}{\partial X^2} + (P + 2QU)\left(\frac{\partial U}{\partial X}\right)^2 - H_1\frac{\partial^4 U}{\partial X^4} + H_2\frac{\partial^4 U}{\partial X^2 \partial T^2}, \qquad (1)$$

```
def ibioD(tulemusx,tvektorx,n): # inverse transfrom of results with H2 uttxx dimensionless EQ
    i = 0 ; um_t_reaalne = [] ; aprox_kiirus = []
    temp2 = [] ; ru = [] ; fft_r = [] # siirde jaoks
    temp3 = [] ; rut = [] ; fft_rt = [] # kiiruse jaoks
    omega = oomega(n)
    while i<len(tvektorx): # transfrom into real space
        temp2 = tulemusx[i][:n] # first part (space)
        temp3 = tulemusx[i][n:] # speed
        fft_r = fft(temp2,n,-1) # r forward
        fft_rt = fft(temp3,n,-1)
        r_ajutine = numpy.arange(n,dtype=numpy.complex128) #
        rt_ajutine = numpy.arange(n,dtype=numpy.complex128) #arange(n,dtype='complex128')
        k = 0
        while k < n: # index 0 kuni n-1. calculation to inverse transform
            r_ajutine[k] = fft_r[k]/(1+HH2*omega[k]**2)
            rt_ajutine[k] = fft_rt[k]/(1+HH2*omega[k]**2)
            k = k+1
        ru = ifft(r_ajutine).real # end of inverse transform
        rut = ifft(rt_ajutine).real
        um_t_reaalne.append(ru) # space [row = coordinate, column = time]
        aprox_kiirus.append(rut)
        i=i+1
    return um_t_reaalne,aprox_kiirus


def Biosech2D(Ao,Bo,x,loik,c,HH2): # Sech**2 initial condition
    xx = x-loik*numpy.pi #profile shift
    u0 = Ao/(cosh(Bo*xx))**2
    u0x = diff(u0,period=2*numpy.pi*loik); #speed -c*u0x
    u0xx = diff(u0x,period=2*numpy.pi*loik);# second space derivative
    yx[:n] = u0-HH2*u0xx # space transformed
    yx[n:] = -c*u0x; #speed;
    return yx.real


def Bio2D(t,yx): #Improved HJ equation, dimensionless form
    nn = len(yx)/2
```

```
    omega = oomega(nn)
    temp2 = [] ; ru = [] ; fft_r = []
    temp2 = yx[:nn] #first part
    fft_r = fft(temp2,nn,-1) #r forward transform
    r_ajutine = numpy.arange(nn,dtype=numpy.complex128) # complex array 'D'
    kkk = 0
    while kkk < nn: #index 0 to n-1.
        r_ajutine[kkk] = fft_r[kkk]/(1+HH2*omega[kkk]**2)
        kkk = kkk+1
    ru = ifft(r_ajutine).real #inverse transform end
    rr = ru*ru  #nonlinear
    v = yx[nn:] #speed or dr/dt, y[n] to y[2n-1]
    x_x = diff(ru,1,period=2*numpy.pi*loik)   # first derivative
    x2 = x_x*x_x                              # square of first derivative
    x_xx = diff(ru,2,period=2*numpy.pi*loik)  # second derivative
    x_xxxx = diff(ru,4,period=2*numpy.pi*loik)# fourth derivative
    ytx[:nn] = v #du/dt=v
    ytx[nn:] = x_xx + PP * ru * x_xx + QQ * rr * x_xx + PP * x2 + 2 * QQ * ru * x2 - HH * x_xxxx
    return ytx

# example set of normalized parameters
co = 1; co2 = co * co; rho0 = 1; l = 1; # geometric / material param
c = 1 + 0.1; gamma2 = (c*c)/co2
p = 0.05; q = 0.075; # nonlinear param
h2 = 0.05; h1 = gamma2*h2; # dispersion param
Bo = 1/8; Ao = 1; # initial pulse param

# dimensionless P = p * roo_0 / co2; Q = q * roo_0**2 / co2; H = h / (co2 * l)
PP = p * rho0 / co2; QQ = q * rho0**2 / co2; HH = h1 /(co2 * l**2); HH2 = h2 /(l**2);

yx = Biosech2D(Ao,Bo,x,loik,c,HH2); tyyp='sech2';
# ---------- integration of EQ -----------------------------------
t0 = 0.0; aeg_alg = clock() #initial time, floating point number
t_end = narv*dt #end time
tvektor1 = []; tvektor2 = [] ; tvektor3 = [] ; tulemus1 = []; tulemus2 = [];  tulemus3 = [];
tvektor4 = [] ; tulemus4 = []; tvektor5 = [] ; tulemus5 = []; tvektorx = []; tulemusx = [];
runnerx = ode(Bio2D) #for dimensionless form!
runnerx.set_integrator('vode',nsteps=1e7,rtol=1e-10,atol=1e-12);
runnerx.set_initial_value(yx,t0);
while runnerx.successful() and runnerx.t < t_end: #-h uxxxx - H2 uxxtt
                  tvektorx.append(runnerx.t);
                  tulemusx.append(runnerx.y);
                  print('z ',runnerx.t);
                  runnerx.integrate(runnerx.t+dt);
(um_t_reaalne,aproxkiirus) = ibioD(tulemusx,tvektorx,n) # inverse transform, dimensionless form
umx = transpose(um_t_reaalne); #umtx = transpose(aproxkiirus) H2 member
# and here you can save the results into Matlab
```

Linux might need first few lines of your Python script to be something like that:

```
#coding=iso-8859-15
#/usr/bin/env python
```

1.2 Visualising and analysing the solutions

Getting the model equations solved is the easy part. Real trick is to make some sense of the stuff that popped out. For that the data needs to be analysed and presented somehow in human understandable format. Often this is some-kind of graphical presentation of data.

The simplest form of visualization is just plotting the 1D waveprofile at some time which is interesting for some reason. While this is simple undertaking in principle there are few tricks

in here for producing graphs that are suitable for publication. Few notes that can make the life simpler in the long run.

- Always write a script. Store that script in some logical location so you can find it a while later. Do a separate script for each separate graph. It takes a little longer at first but really helps down the road when the supervisor or editor or reviewer asks "X" months later to change something.
- By default (that means as Matlab and Python using matplotlib save them) dashes and dots in .eps format files which are usually used for scientific publication are ugly. Open the file in text editor, find field "0 cap" and change it to "1 cap". Leave this process for the last as if you need to change something file gets overwritten (you did write a script for that right?)
- If the amount of data or number of data files is significant it is advisable to implement scripts for speeding up the process of "getting some kind of picture". Look at example provided which generates just quick .jpg format plots of all fittingly named files in a directory so you can flip through them in more or less sane manner.
- If the data has more than 2 dimensions (for example, numerical experiment results where you are trying to determine what is the effect of 7 different parameters in 1D model equation) be extra careful, try to somehow group the dimensions based on some kind of shared characteristics. Make sure all the parameters you are looking are really independent, if they are not reduce the degrees of freedom to the minimal possible set. If it is not clear from the picture what you are trying to show most people will not understand what you are trying to show from long winded explanation.
- Colors are useful and help to make the graph readable in electronic formats but for all practical purposes you are limited to maximum 4 different lines per 2D plot. Journals are printed in black and white and red and blue are exactly the same when printed in black and white. The 4 different lines are line types in Matlab, (solid, dashed, dash-dotted and dotted lines).
- It can be useful to use "pseudo 3D". See the "timeslice" plot example.
- For conferences or other presentations a good animation might be better than dozen graphs for explaining some kind of dynamic process.
- Best scripts are hand written. Failing that you can auto-generate a script for getting figure which is initially done manually in Matlab figure environment .. I think .. somehow. Auto generated scripts can contain a lot of noise and can be quite hard to dig through if you do need to change something later (you will most likely).

Example 1: Matlab script for generating .jpg files from a directory of data-files for getting a overview of what is there. In here and following example the "um" is data array holding the data, first index is space and second index is time coordinate.

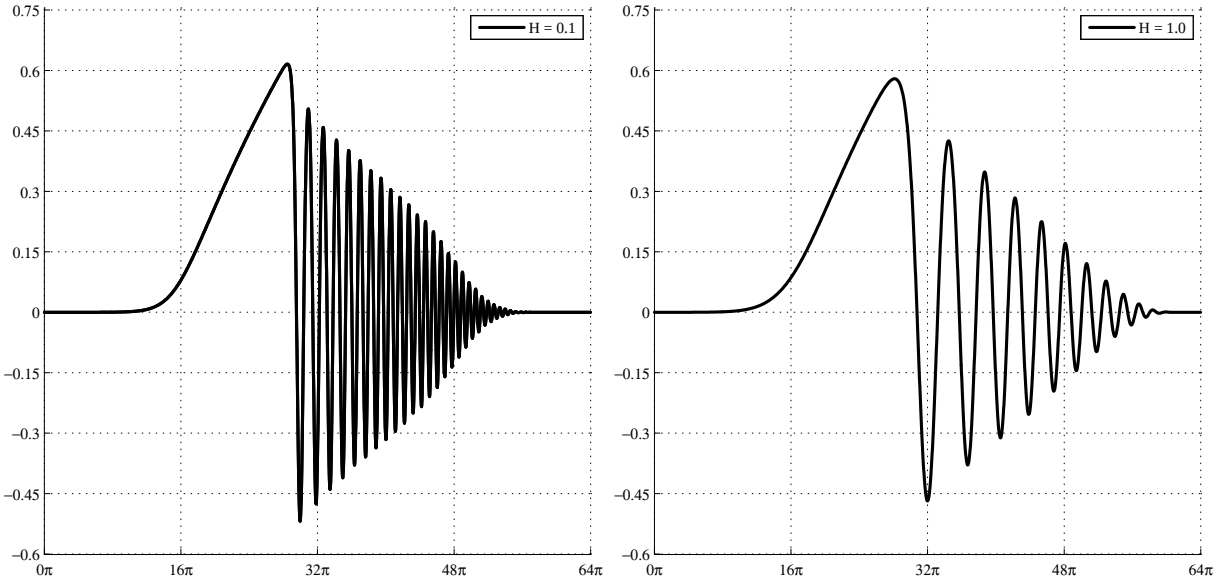
```
clear; %esimene joon
cd('D:\Teadus\Tallinn2014\Ettekande_data\Large') % dir with data
poz = [1           1        1920        1088]; %you do not need that
dataFiles = dir('HN_IUTAM_Hvar09*.mat'); %find suitable files
numfiles = length(dataFiles); %number of fitting files
for s=1:numfiles %
    cd('D:\Teadus\Tallinn2014\Ettekande_data\Large')
    load(sprintf('%s',dataFiles(s,1).name)); %load datafile
    figure(s); hold on;
    plot(um(:,1),'linestyle','-','color','k','linewidth',2) %T=1
```

```
    plot(um(:,5902),'linestyle',':','color','r','linewidth',2) %T=5902
    ylabel('{\it U}','FontSize',16,'FontName','Times New Roman');
    xlabel('\it X','FontSize',16,'FontName','Times New Roman');
    axis tight; grid on;
    cd('D:\Teadus\Tallinn2014\pildid\Large') %place where to put pictures
    print(s,sprintf('%s',dataFiles(s,1).name),'-djpeg');
end
```

"print" in script is preferable to the "save as" in Matlab figure environment as it "freezes" the font-size into the figure. One should be writing scripts for everything anyway.

Example 2: Generating the figures for a journal publication. Was enchanted before final submissions with the "0 cap to 1 cap" macro.

```
% Tallinn 2014 konverents artikkel joonis 2
% Gamma 1.1 H changes two fig side by side
clear; %esimene joon
cd('D:\Teadus\Tallinn2014\Ettekande_data\Large') % kataloog kus on datafailid
poz = [1           1        1920        1088];
moment2=1070;
figure(21); hold on;
load('Fig2a.mat');
[n m]=size(um); n2=.5*n;
plot(x(1:n2),um(1:n2,moment2),'k','clipping','off','linewidth',2,'linestyle','-');
set(gca,'XTick',0:16*pi:64*pi,'XTickLabel',{'0p','16p','32p','48p','64p'},'FontName','Symbol');
set(gca,'YTick',-0.6:0.15:0.75,'YTickLabel',{'-0.6','-0.45','-0.3','-0.15','0','0.15','0.3',...
'0.45','0.6','0.75'},'FontName','Symbol');
axis([0 64*pi+0.01 -0.601 0.755]);
l1=legend('H = 0.1'); set(l1,'Location','NorthEast'); set(l1,'FontName','Times');
grid on; set(21,'pos',poz); pbaspect([1 1 1])
cd('D:\Teadus\Tallinn2014\Artikkel')
print(21,sprintf('Fig21_Gamma2_%d_H1_%d_H2_%d_P_%d_Q_%d.eps',gamma2,H1,H2,PP,QQ),'-dpsc2');

cd('D:\Teadus\Tallinn2014\Ettekande_data\Large')
figure(22); hold on;
load('Fig2b.mat');
[n m]=size(um); n2=.5*n;
plot(x(1:n2),um(1:n2,moment2),'k','clipping','off','linewidth',2,'linestyle','-');
set(gca,'XTick',0:16*pi:64*pi,'XTickLabel',{'0p','16p','32p','48p','64p'},'FontName','Symbol');
set(gca,'YTick',-0.6:0.15:0.75,'YTickLabel',{'-0.6','-0.45','-0.3','-0.15','0','0.15','0.3',...
'0.45','0.6','0.75'},'FontName','Symbol');
axis([0 64*pi+0.01 -0.601 0.755]);
l1=legend('H = 1.0'); set(l1,'Location','NorthEast'); set(l1,'FontName','Times');
grid on; set(22,'pos',poz); pbaspect([1 1 1])
cd('D:\Teadus\Tallinn2014\Artikkel')
print(22,sprintf('Fig22_Gamma2_%d_H1_%d_H2_%d_P_%d_Q_%d.eps',gamma2,H1,H2,PP,QQ),'-dpsc2');
```

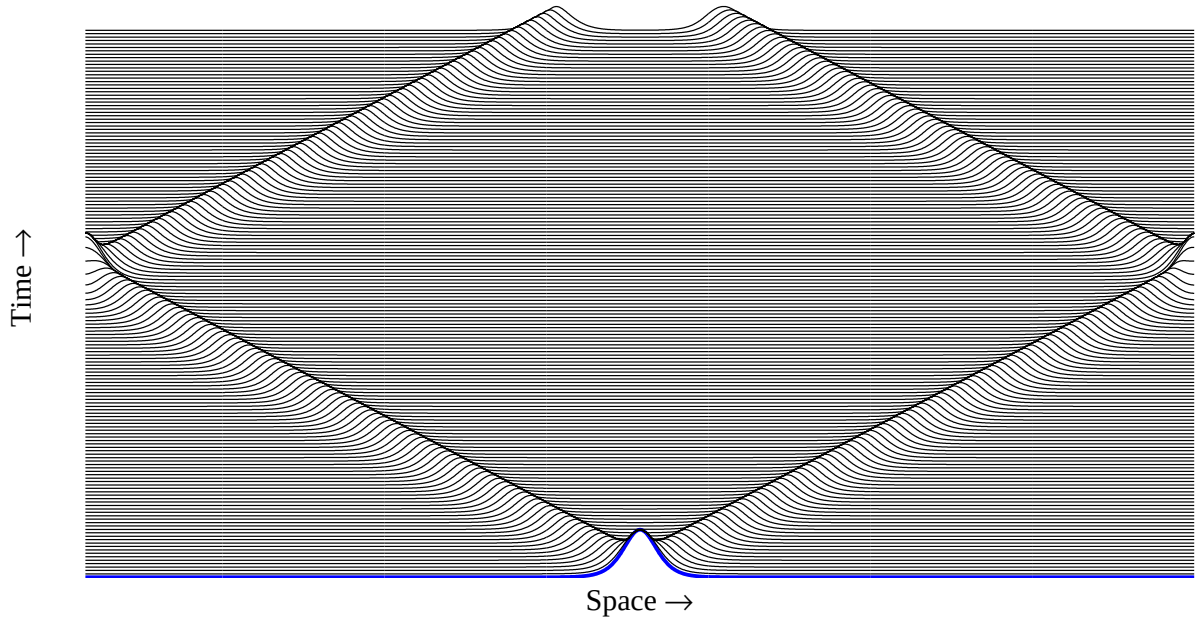


Note how most of the lines are more or less the same. After you have established the first few scripts for proper visualization the following ones will go much easier.

Example 3: Timeslice plot

```
% EC540 profiilid
clear; %esimene joon
cd('F:\Teadus\WAVE13\USA2013') % datafiles
poz = [1           1        1920        1088];
file = 'USA_II_GA_0.2_G1_0.4_c_0.0_Ao_1_n_2048_12pi_32_A_20_B_25.0_C_3.2_D_10_N_50_M_500.mat';
load(file); %load datafile
um = um4(:,1:15:2401); %drawing every 15th line in time of array um4
ku = 12; %viewing angle
figure(1); set(1,'position',[poz]);

ij=[];
clf,hold off
[n,m]=size(um);
if 9==0,N=64;   %  increase num of gridpoints if needed
    if n<N,
        nd=N/n;
        dx=(x(2)-x(1))/nd; x=dx*[0:N-1]';
        U=fft(um);
        um=nd*ifft([U(1:.5*n,:);zeros((nd-1)*n,m);U(.5*n+1:n,:)]);
        n=N;
    end
end
u0=um(:,1); u=um(:,m);
umax=max(real(u));
ymin=min(real(u0));
ymax=ymin+ku*abs(max(real(u0))-ymin);
am=(ymax-umax)/(m-1);
uum=um+am*ones(n,1)*[0:m-1];
ax=[min(x),max(x),1.05*ymin,1.05*ymax];

axis(ax), axis('off');
hold on

u0=uum(:,1);
h=plot(x,u0,'b-','linewidth',1.5);

for l=2:m
u1=uum(:,l);
```

```
i=find(u1<u0);
if length(i)>=1, u1(i)=u0(i); end
h=plot(x,u1,'k-'); %,'linewidth',1
u0=u1;
end;

hy=ylabel('Time \rightarrow','visible','on','FontSize',12,'FontName','Times');
ylxy=get(hy,'position')
1
hx=xlabel('Space \rightarrow','visible','on','FontSize',12,'FontName','Times');
set(hx,'VerticalAlignment','bottom')
xlxy=get(hx,'position')

if 0
ttx=[setstr(190),setstr(190),setstr(190),setstr(190),setstr(174)];
huh=text(xlxy(1),xlxy(2),ttx,'fontsize',18);  %text(xlxy(1),xlxy(2),'X')
set(huh,'fontname','symbol',...
   'HorizontalAlignment','center',...
   'VerticalAlignment','top')

huh=text(ylxy(1),ylxy(2),ttx,'fontsize',18);
set(huh,'fontname','symbol',...
   'HorizontalAlignment','center',...
   'VerticalAlignment','bottom','Rotation',90)
set(gcf,'paperunits','centimeters','paperposition',[3,9,16,12])
end

pbaspect([1.85 1 1])
print(1,sprintf('TSlice_II_A_%d_B_%d_C_%d_D_%d_N_%d_M_%d.eps',A,B,C,D,Nparam,Mparam),'-dpsc2');
```



In essence timeslice plot is similar in function to contour or pseudocolor plots. The main strength of this plot-style is that it is very intuitive and gives a good overview of solution behaviour at a glance.

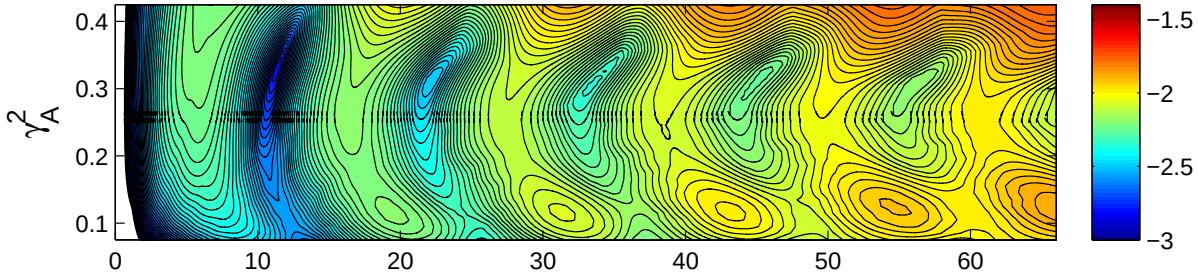
Example 4: Making animation in Matlab. Reasonably fancy set up with four sub plots running in one window in parallel. If you need animation make it early, this can take a while, especially if you do it in high resolution. In older Matlab versions file size limit is 2 GB which gets full really fast if you do not use compression. Proper compression is best done afterwards in some other program. If you run into file size limits it is possible to do animation in pieces and merge pieces together afterwards in some other program.

```
% EC540_presentantsiooni muuvi
```

```
% um1 - approx, um - full eq
% um3 - doublemcro hierarchical um4 - doublemcro concurrent connected
% um5 - doublemcro concurrent intependent
clear
poz = [1           1        1920/2        1088/2];
cd('F:\Teadus\WAVE13\USA2013') % datafiles
dataFiles = dir('USA_II_GA_0.4_G1_0.6_c_0.0_Ao_1_n_2048_l2pi_32_A_20_B_12.5_C_4.8_D_10_N_50_M_500.mat');
numfiles = length(dataFiles); %laetavate failide arv
for s=1:numfiles %
    load(sprintf('%s',dataFiles(s,1).name)); %laeb sisse vajalikud muutujad
    [n m]=size(um); n2=.5*n; %t = length(tv);
    muuvi = [sprintf('Run_II_GA_%d_G1_%d_A_%d_B_%d_C_%d_D_%d_N_%d_M_%d',GA,G1,...
    A,B,C,D,Nparam,Mparam),'.mp4']; %'.avi'
    writerObj = VideoWriter(muuvi,'MPEG-4') %'Uncompressed AVI'
    writerObj.FrameRate = 66; %writerObj.Quality = 100;
    open(writerObj);
    ffs=figure(s);
    set(s,'units','points');
    set(s,'position',[poz]); %hoho = gca;
    for j = 1:m
        subplot(2,2,1);
        plot(x,um(:,j),'r','clipping','off','linewidth',2,'linestyle','-'); hold on
        xlabel('0 \leq {\it X} < 64\pi','FontSize',10,'FontName','Times');
        ylabel('{\it U}','FontSize',10,'FontName','Times');
        set(gca,'XTick',0:8*pi:64*pi,'XTickLabel',{'0p','8p','16p','24p','32p','40p','48p','56p',...
        '64p'},'FontName','Symbol');
        set(gca,'YTick',0:0.1:0.5,'YTickLabel',{'0','0.1','0.2','0.3','0.4','0.5'},'FontName','Symbol');
        axis([0 64*pi+0.01 0 0.505]);
        l1=legend(['FSE T = ',num2str(tv(j),'% 10.2f')],'Location','NorthWest');
        set(l1,'FontName','Times');
        grid on; hold off;
        % --------------------------------
        subplot(2,2,2);
        plot(x,um3(:,j),'k','clipping','off','linewidth',2,'linestyle','-'); hold on
        xlabel('0 \leq {\it X} < 64\pi','FontSize',10,'FontName','Times');
        ylabel('{\it U}','FontSize',10,'FontName','Times');
        set(gca,'XTick',0:8*pi:64*pi,'XTickLabel',{'0p','8p','16p','24p','32p','40p','48p','56p',...
        '64p'},'FontName','Symbol');
        set(gca,'YTick',0:0.1:0.5,'YTickLabel',{'0','0.1','0.2','0.3','0.4','0.5'},'FontName','Symbol');
        axis([0 64*pi+0.01 0 0.505]);
        l3=legend(['HED T = ',num2str(tv(j),'% 10.2f')],'Location','NorthWest');
        set(l3,'FontName','Times');
        grid on; hold off;
        % --------------------------------
        subplot(2,2,3);
        plot(x,um4(:,j),'b','clipping','off','linewidth',2,'linestyle','-'); hold on
        xlabel('0 \leq {\it X} < 64\pi','FontSize',10,'FontName','Times');
        ylabel('{\it U}','FontSize',10,'FontName','Times');
        set(gca,'XTick',0:8*pi:64*pi,'XTickLabel',{'0p','8p','16p','24p','32p','40p','48p','56p',...
        '64p'},'FontName','Symbol');
        set(gca,'YTick',0:0.1:0.5,'YTickLabel',{'0','0.1','0.2','0.3','0.4','0.5'},'FontName','Symbol');
        axis([0 64*pi+0.01 0 0.505]);
        l4=legend(['COC T = ',num2str(tv(j),'% 10.2f')],'Location','NorthWest');
        set(l4,'FontName','Times');
        grid on; hold off;
        % --------------------------------
        subplot(2,2,4);
        plot(x,um5(:,j),'g','clipping','off','linewidth',2,'linestyle','-'); hold on
        xlabel('0 \leq {\it X} < 64\pi','FontSize',10,'FontName','Times');
        ylabel('{\it U}','FontSize',10,'FontName','Times');
        set(gca,'XTick',0:8*pi:64*pi,'XTickLabel',{'0p','8p','16p','24p','32p','40p','48p',...
        '56p','64p'},'FontName','Symbol');
        set(gca,'YTick',0:0.1:0.5,'YTickLabel',{'0','0.1','0.2','0.3','0.4','0.5'},'FontName','Symbol');
```

```
        axis([0 64*pi+0.01 0 0.505]);
        l5=legend(['COI T = ',num2str(tv(j),'% 10.2f')],'Location','NorthWest');
        set(l5,'FontName','Times');
        grid on; hold off;
        % -------
        F(j) = getframe(ffs);
        writeVideo(writerObj,F(j));
        pause(0.02); %oota 20 millisekundit
        clear F; hold off;
    end
    close(writerObj);
end4
```

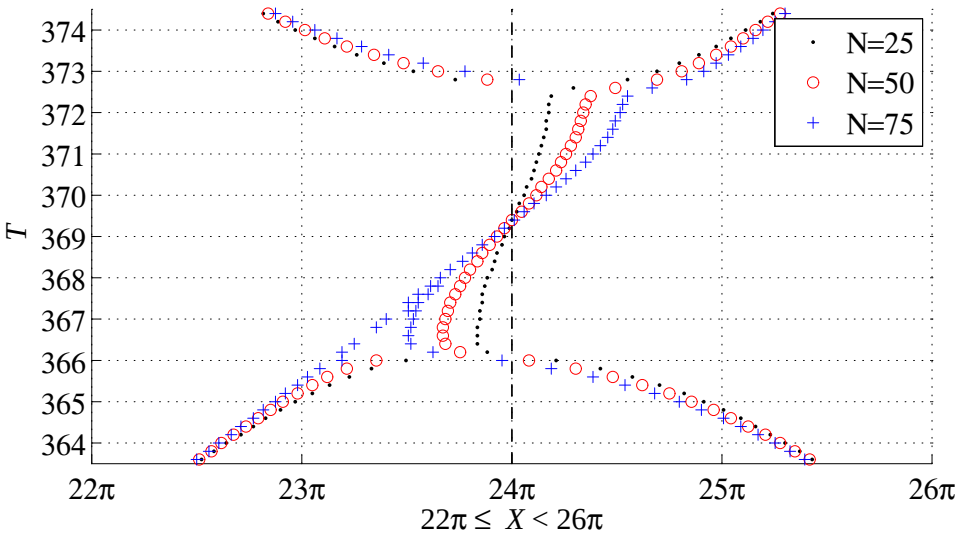
One does not need to avoid colour all the time. In particular in presentations colour is useful aspect for making graphs easier to read, however, always have a plan how you can present your data in black and white if you will need to write something about the presented things afterwards.



Example filled contour plot. Difference between two solutions in time, logarithmic scale. Plotted in axis of combined parameter γ_A^2 and time.

Important note about using contour plots (and any kind of isolines in general) in Matlab. The algorithm looks only at closest 4 orthogonal neighbours so if you have some kind of ridge along the "diagonal" (the other 4 neighbours of the cell) you will get artificial visual oscillations which are just a visual artefact! Before you go off and claim to have found some kind of new never before seen effect make sure it's not an artefact of your visualisation algorithm.

Another alternative to contour or pseudocolor plot is tracking the wave peak which can be useful for highlighting some effects in a format that is visible when printed in black and white as well. Example:



1.2.1 From Python to Matlab

Saving in Python in Matlab format can be useful if you use Matlab for analysing and visualizing the results. There are free alternatives (Matlab is relatively expensive program) like using matplotlib in Python environment or GNU Octave which is open-source Matlab alternative.

```
from scipy.io import savemat
% solving the equations is here
fn = 'HN_IUTAM_Hvar11_H_%s.mat'%(HH2)
sonaraamat = dict([('um', (umx)), ('tv', (transpose(tvektorx))), ('x', x),\
    ('co', co), ('loik', loik), ('Ao', Ao), ('Bo', Bo), ('oo', omega),\
    ('arvutusaeg_sekund', aeg_arvutus), ('p', p), ('q', q),\
    ('PP', PP), ('QQ', QQ),('H1', HH), ('H2', HH2), ('gamma2', gamma2)]);
savemat(fn,sonaraamat);
```

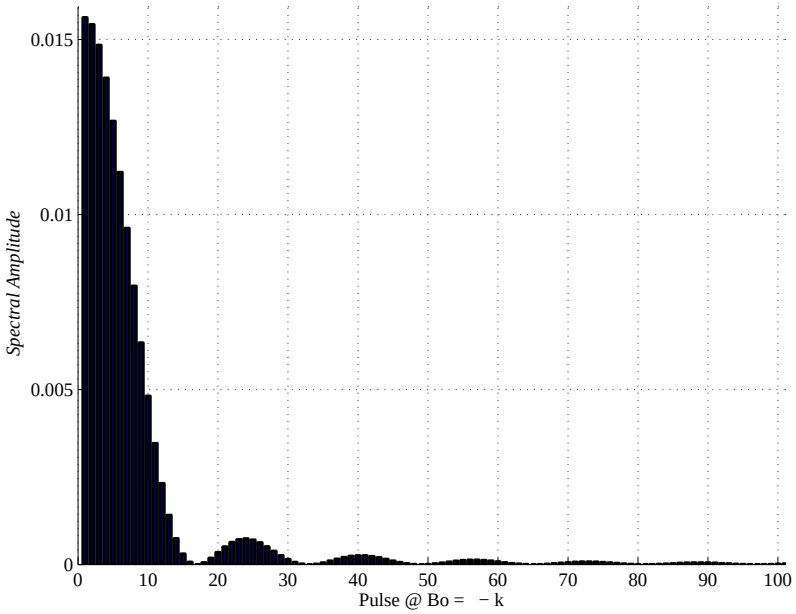
The dictonary contains pairs of variable name in Matlab style save file and the value of that variable.

1.2.2 Spectral analysis

Example: Spectral amplitude histogram in Matlab

```
ina = 0; TT=5902;
[n,m]=size(um);
komponent=n/2;
Uavektor = zeros(komponent);
figure(100); hold on
for i=1:komponent
    Ua=fft(um);
    Ua=2*abs(Ua(i,TT))/n;  %presuming that ina(end)<n2
    Ua=Ua.^2;
    Uavektor(i) = Ua;
end
bar(Uavektor(:,1),'LineStyle','-','LineWidth',2,'BarWidth',0.6); axis tight; grid on;
xlabel(sprintf('Pulse @ Bo = %s - k',Bo),'FontSize',12,'FontName','Times New Roman');
ylabel('{\it Spectral Amplitude}','FontSize',12,'FontName','Times New Roman');
set(gca,'FontSize',12,'FontName','Times New Roman');
axx=[0,101,0, max(max(Uavektor))+0.02*max(max(Uavektor))];
axis(axx)
print(100,'Spekter_alg_n4096.eps','-dpsc2');
```

Resulting in the following graph for rectangular function pulse at the initial time-step at $n = 8192$ (meaning that there is 4096 harmonics in the Fourier series altogether)

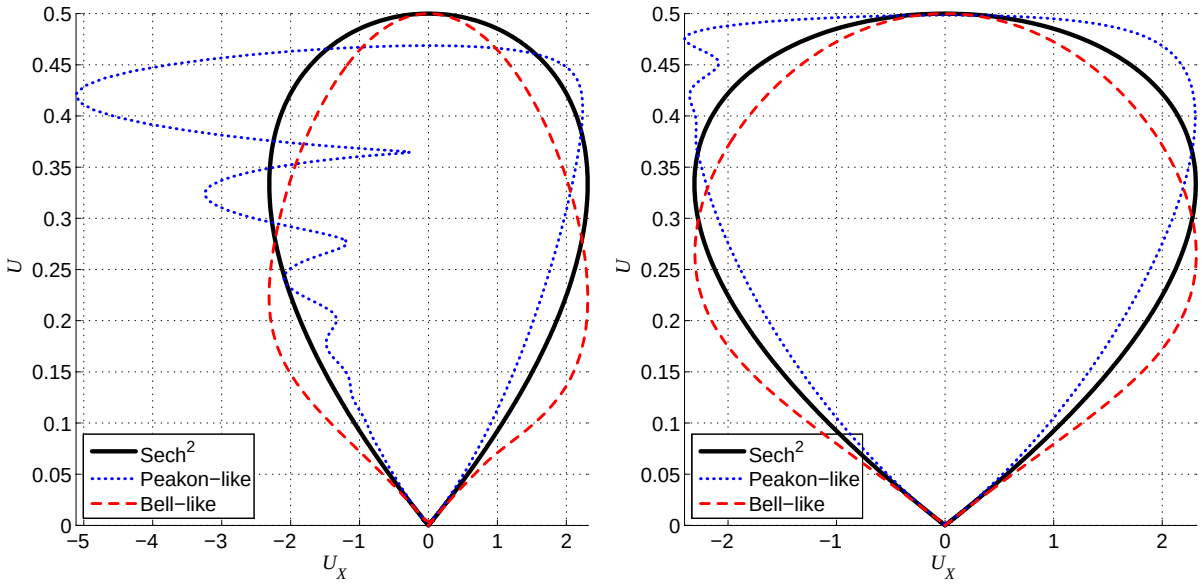


Depending on what is under investigation it is often reasonable to look at spectral amplitudes in the logarithmic scale.

This is just one example how it is possible to take look at the spectrum of your waveprofile. It is advisable to go over the cited textbook PSM section which contains the analytical formulas.

1.2.3 Phase plots

Looking at just plain waveprofile plot at given time moment it can be hard to spot small changes in the waveprofile shape. One possibility of highlight (or amplify) small changes is phase plot which is in essence plotting the waveprofile in a little unusual axis. For 1D wave plot the reasonable axis are U and U_X, however these are not the only options. Couple examples:



References

[1] A. Salupere. The pseudospectral method and discrete spectral analysis. In Ewald Quak and Tarmo Soomere, editors, *Appl. Wave Math.*, pages 301–334, Berlin, 2009. Springer.

1 2D Pseudospectral example

Kert Tamm, kert@ioc.ee and Mart Ratas, Pearu Peterson

Example of Pseudospectral method (PSM) implementation in 2D based on a Python code from the e-mail correspondence between Mart Ratas and Pearu Peterson of which I was also part of. Python script has very minor editing by Kert Tamm to make it run without errors under Python 3.x. Matlab script for reshaping and visualizing by Kert Tamm.

The example is based on the classical heat equation in 2D

$$u_t - \alpha \left(u_{xx} + u_{yy} + u_{zz} \right) = 0, \tag{1}$$

for 2D a u_{zz} is just absent. Subscript denotes partial differentiation.

The Python script:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 12 17:51:42 2015
PSM HEAT EQUATION
u_t=a*(u_xx+u_yy)
12.04.15 algus
@author: Mart
"""

#siin paneme paika vajalikud parameetrid

punktide_arv=256
salv=.01    #kui tiheda sammuga salvestan
tf=0.1       #lppaeg
atol=1e-12
rtol=1e-10
nsteps=1e7
integrator='vode'

#momg vajalikud raamatukogud
import numpy, scipy.io

def psm_start(punktide_arv,salv,tf,atol,rtol):
    dx=(1*2*numpy.pi/punktide_arv)
    #sisuliselt omega, et tuletisi vtta
    abi=numpy.hstack((numpy.arange(punktide_arv/2),numpy.arange(-punktide_arv/2,0)))
    suurK=abi
    for i in range(punktide_arv-1):
        suurK=numpy.vstack((suurK,abi))

    #omega transponeeritud, et tuletisi vtta y'st
    suurL=suurK.swapaxes(0,1)

    #print 'max,min',numpy.max(suurKP),numpy.min(suurKP)
    x=dx*numpy.arange(-punktide_arv/2,punktide_arv/2)
    y=x

    xx,yy=numpy.meshgrid(x,y)
    u0=numpy.exp(-(xx**2+yy**2)*25.0)

    #nd siis integreeerima!
    import scipy.integrate

    print('integreerin ',integrator,"'ga", u0.shape)
    solver=scipy.integrate.ode(funktsioon).set_integrator(integrator,nsteps=nsteps,atol=atol, rtol=rtol)
```

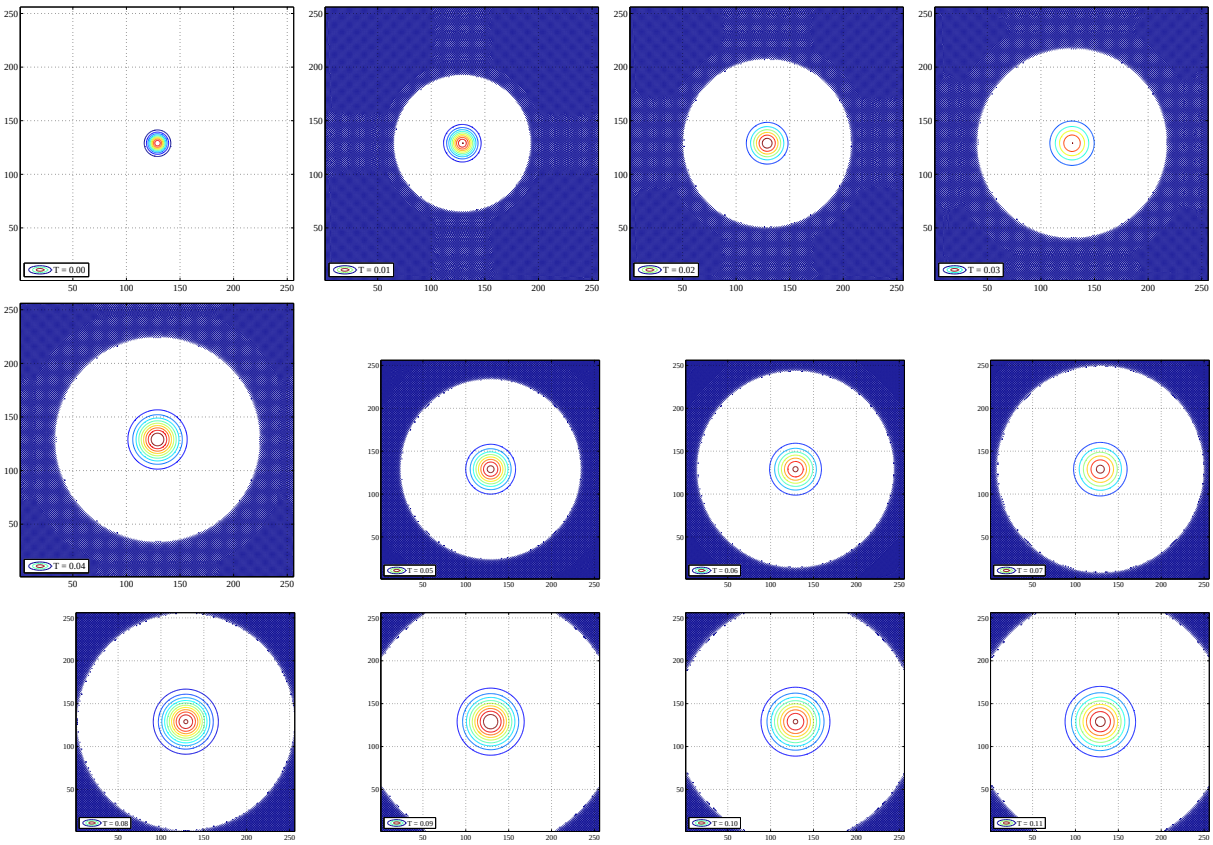
```
    u0 = u0.flatten ()
    solver.set_initial_value(u0, 0).set_f_params(suurK,suurL)
    lahend=[u0.copy()]
    tv=[0,]
    counter=0
    while solver.t< tf:#solver.successful() and solver.t < tf:
        counter+=1
        solver.integrate(solver.t + salv)
        print(numpy.min(solver.y), numpy.max(solver.y))

        ucurrent=solver.y
        print(numpy.shape(ucurrent), numpy.max(numpy.abs(ucurrent-u0)))

        #proovisin!

        lahend.append(ucurrent)
        #print numpy.min(ucurrent), numpy.max(ucurrent)
        tv.append(solver.t)
        print('arvutan, aeg: ',solver.t,solver.successful())
        if counter-1>tf/float(salv):
            break

    scipy.io.savemat('heat_eq_test_12_04',{'u0':u0,'suurK':suurK,'lahend':lahend,'tv':tv,'x':x\
,'y':y,'punktide_arv':punktide_arv})

def funktsioon(t,xxyy,suurK,suurL):
    #kasuta seda integreerimiseks!
    import numpy
    a=1
    xxyy = numpy.reshape( xxyy, (punktide_arv,)*2)

    #print 'SIIN!!!'
    #print '============================================================='
    uxx=numpy.real(numpy.fft.ifft(-suurK**2*numpy.fft.fft(xxyy,axis=1),axis=1))
    uyy=numpy.real(numpy.fft.ifft(-suurL**2*numpy.fft.fft(xxyy,axis=0),axis=0))

    tul=a*(uxx+uyy)
    #print 'sain tulemuse:', tul.shape
    #print max(tul),min(tul)
    return tul.flatten ()

psm_start(punktide_arv,salv,tf,atol,rtol)
```

Do note that this script gives out a vector, not matrix as one would expect for vizualisation, so the visualising Matlab script reshapes the result vector into correct format.

```
% 2D heat EQ visualization
clear
fsize = 12; %font size
cd('D:\Teadus\Python') %kataloog kus on datafail
load('heat_eq_test_12_04.mat'); % failinimi
U02D=reshape(u0,[punktide_arv,punktide_arv]); % initial condition
for i=1:length(tv)
    Lah2D{i,1}=reshape(lahend(i,:),[punktide_arv,punktide_arv]);
end
% plotting
for i=1:length(tv)
    hh=figure(i)
    contour(Lah2D{i,1})
    l1=legend(['T = ',num2str(tv(i),'% 10.2f')],'Location','SouthWest');
```

```
    set(l1,'FontName','Times','FontSize',fsize);
    set(gca,'FontName','Times','FontSize',fsize); %axis to correct font
    grid on; pbaspect([1 1 1]); %grid and aspect ratio
    print(hh,sprintf('%d_T_%d.eps',i,tv(i)),'-dpsc2'); %plot eps
end
```

Te resulting plots should look by default something like this:



Please ignore the different size of the graphs. This is some kind of a LaTeX issue as the resulting .eps files are all of same size from the visualisation script.

Few things to note when using Matlab contour and contourf functions. (1) The blue area around the edges is very low amplitude noise which contour plot has picked up when it sets the levels automatically - for a clear picture one would need to set the isolines manually and in a such way that they are the same for all graphs one intends to compare. (2) Matlab contour plots can and do create artefacts which are not present in the underlying data as the algorithm only takes into account the 4 perpendicular neighbours of the data-point ignoring the diagonal neighbours. So if you have a maxima running on the diagonal of the matrix you will get a row of "islands" on the diagonal instead of a single continuous ridge present in the data.

Limit of wavenumbers in Fourier Pseudospectral Method considering available numerical precision

Martin Lints
TUT Institute of Cybernetics

May 11, 2015

1 Introduction

Pseudospectral methods are widely used to calculate partial differential equations numerically, often for equations which require spatial derivatives of high orders [1]. These equations can require wavenumber filtering to suppress the contribution of high wavenumbers [2] for the solution to remain stable in computations in danger of shock conditions or discontinuities [3].

This paper discusses an alternate source of unstability due to the problem of finite accuracy of computers which can also destroy the solution by multiplying the energy in high wavenumbers by a large number. It will be shown that the numerical discretization must be limited in order for the computation to not have unphysical oscillations produced by the limited precision of the computer. For this is useful to know the point where the numerical precision does not support any additional information.

Because computers work on binary numbers of finite length in each register, round-off and representation errors occur for floating point numbers. Moreover floating point operations do not necessarily behave like arithmetic operations [4]. Dahlquist [5] has pointed out that the floating point addition and multiplication are commutative but not associative nor distributive. In short we only get a finite precision when using traditional programming languages in calculations. It is possible to extend the precision by using, for example, quadruple precision numbers built now into some compilers or software packages which allow arbitrary precision. This comes with additional computational cost per operation, because the number does not "fit" into the CPU and this computational expense needs to be carefully considered.

Modern CPU-s mostly use 64-bit arithmetic which, in case of IEEE 754 "double precision" floats, comprise of 1 sign bit, 11 exponent bits and 52 significand bits. These 52 significand bits can represent a decimal number with ~ 16 significant decimal places. The "extended precision" of 80-bit and "quadruple precision" of 128-bit numbers and higher has lately become available in some Fortran and C/C++ compilers (gcc), or as external libraries which can be used if more precision is required. The effect of various precision computations on the end result will be shown in the following work.

The numerical codes for this paper were written in C using the FFTW3 library [6] and compiled with Gnu C Compiler (gcc) version 4.8.2 with its quadmath library for __float128 quad precision support.

2 The application of Fourier Pseudospectral Method

In the Fourier Pseudospectral Method, the spatial derivatives of a periodic function are found by approximating it with a Fourier series. The theory is well known, so only the main steps of the application are shown in this section. As summarized by Salupere [1], it is a global method approximating a function as a sum of smooth basis functions $\Phi_k(x)$:

$$u(x) \approx \sum_{k=0}^{N} a_k \Phi_k(x). \tag{1}$$

If the function $u(x)$ is periodic then we can choose trigonometric functions as a basis and use Fast Fourier Transform (FFT) algorithms to do the work efficiently. This function, given in an interval $0 \leq x \leq 2\pi$ and space grid is composed of N points, has a Discrete Fourier Transform (DFT)

$$U(k,t) = Fu = \sum_{j=0}^{N-1} u(j\Delta x, t) \exp\left(-\frac{2\pi i j k}{N}\right), \tag{2}$$

and inverse DFT(IDFT)

$$u(j\Delta x, t) = F^{-1}U = \frac{1}{N} \sum_{k} U(k,t) \exp\left(\frac{2\pi i j k}{N}\right), \tag{3}$$

where i is imaginary unit, and wavenumbers are

$$k = 0, \pm 1, \pm 2, \ldots, \pm(N/2-1), -N/2. \tag{4}$$

The derivatives of the approximation of the function $u(x)$ in Eq. (1) are given by only the derivatives of $\Phi_k(x)$, as the coefficients a_k do not depend on x. Differentiating the IDFT Eq. (3)

$$\frac{\partial u(x,t)}{\partial x} \approx \frac{\partial u(j\Delta x, t)}{\partial x} = \frac{\partial F^{-1}U}{\partial x} = \frac{1}{N} \sum_{k} (ik) U(k,t) \exp\left(\frac{2\pi i j k}{N}\right), \tag{5}$$

where $\Delta x = 2\pi/N$. Alternatively, if the space period is $2m\pi$, then $\Delta x = 2m\pi/N$ and the quantities k/m are used instead. Therefore the differentiation $\frac{\partial}{\partial x}$ in the original space transforms to multiplication by ik/m in Fourier space in the following way

$$\frac{\partial^n u(x,t)}{\partial x^n} = F^{-1}\left[\left(\frac{ik}{m}\right)^n Fu\right]. \tag{6}$$

3 Example with numerical derivatives of $\sin(x)$

Lets look at the range where $x = [0, 2\pi)$ (or in other words $m = 1$ in Eq. (6)). This range is discretisized with 2048 points, allowing the Fourier transform to contain very high wavenumbers. Various derivatives of $\sin(x)$ are examined. The sinusoid has infinitely many derivatives, all of which are sinusoidal functions and should not have any energy in wavenumbers other than $|k| = 1$, making the numerical result easy to compare with analytical result and exposing the numerical inaccuracies.

The derivatives of $\sin(x)$ are found with Eq. (6), by using DFT and IDFT from FFTW library [6]. It is done for several different precisions: i) 32-bit single precision; ii) 64-bit double precision; iii) 80-bit extended precision; and iv) 128-bit quadruple precision. It is observed that FFT routines usually have a precision of 2-3 magnitudes of order above

machine zero at any given precision (for a derivation of precise error bounds, see [7] and the accuracy benchmarks for FFTW [8]). This means that when applying a FFT to a sinusoid (where only $|k| = 1$ wavenumbers should be nonzero) , the wavenumbers $|k| \neq 1$ oscillate near the maximum representable precision, seen from the red lines in Figs. 1(b), 2(b) and 3(b): i) $\sim 5 - 6$ decimal places for 32-bit single precision; ii) $\sim 13 - 14$ decimal places for 64-bit double precision; iii) $\sim 16 - 17$ decimal places for 80-bit extended precision; iv) $\sim 32 - 33$ decimal places for 128-bit quadruple precision. The red lines in the aforementioned figures show a bit higher precision because for the plotting the wavenumbers have been normalized by $N/2$ in order for the lines to start at magnitude of 1.

In pseudospectral differentiation with high enough derivatives and wavenumbers, the oscillations can and will become apparent. This happens due to the multiplication by the wavenumber vector $(ik)^n$ (where n is the order of the derivative in Eq. (6)), shown by dotted lines in Figs. 1(b), 2(b) and 3(b). Multiplying the numerical oscillations on the limit of the precision with high enough wavenumbers can bring the contribution of the high wavenumber modes to close to the same magnitude as the useful information contained in wavenumbers $|k| = 1$ of the sinusoid. This is shown by green and blue lines in the aforementioned figures.

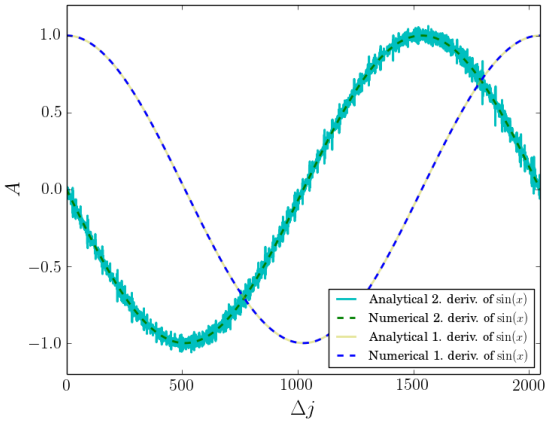
Even though the precision offered from the FFT seems enough for most cases, when taking a high spatial derivative of datasets with thousands of space-grid points the Eq. (6) leads to multiplication by a wavenumber $(ik)^n$ that can have very large numbers. For example when taking a 5-th derivative of $\sin(x)$ using pseudospectral method with double precision FFT, the highest wavenumber in the power of 5 (disregarding the imaginary unit i) would be $(N/2)^5 = 1024^5 = 1.126 \cdot 10^{15}$ which will bring numerical error oscillations (magnitude of $\sim 10^{14} - 10^{15}$) to the visible range, seen by the green and blue lines in Figs. 1(b), 2(b) and 3(b). The loss of accuracy is shown for various precisions in Figs. 1(a), 2(a) and 3(a). After that point any further derivatives will show just noise.

Expanding this further and knowing the desired precision after the differentiation, it is possible to work out the upper limit of the number of spatial points N. For example, knowing that FFT in double precision is precise to $\sim 10^{14}$ and supposing that fifth spatial derivative is needed with accuracy $\sim 10^6$, then the maximum elements of the wavenumber vector can be of magnitude $\sim 10^8$. Since $(N/2)^5 < 10^8 \Rightarrow N < 80$. Figure 4 shows this relation in the clearest way. In practical calculation, since the $|k| = 1$ wavenumber has magnitude of 10^3, even $N < 200$ is suitable and gives a maximum difference of $1.02 \cdot 10^6$ between numerical and analytical 5-th derivative of $\sin(x)$, but in general this cannot be assumed.

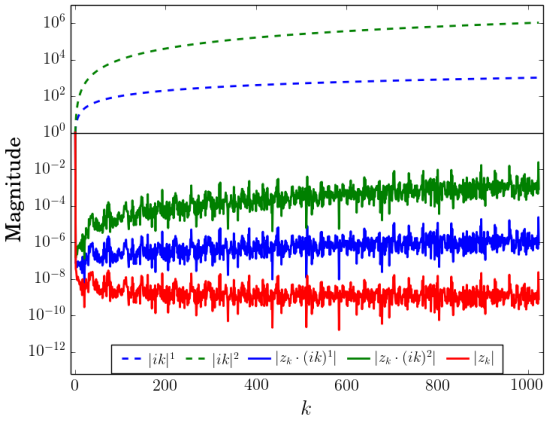
3.1 Practical considerations

This exercise, as also most of the simulations, deal with real-valued data in the original vector $u(x)$. It makes sense to use the real-to-complex transform as DFT and complex-to-real as IDFT such as such as `fftw_plan_dft_r2c_1d` and `fftw_plan_dft_c2r_1d`. This should bring a speedup of factor of two compared to using complex-to-complex transforms.

Filtering is often used in pseudospectral methods for various purposes: suppressing the influence of higher harmonics [1], preventing unwanted transfer of energy into higher modes, to counteract aliasing errors or to improve the convergence in case of discontinuities [9]. In these cases, only the wavenumbers that passed through low-pass filter should be considered as the wavenumbers which would get multiplied by elements of vector $(ik)^n$. However, it might not be advisable to use the filter for the sole purpose of increasing the discretization of the spatial length N and compensating for it with a filter, because the maximum largest frequency that can be represented this way by the pseudospectral method will still remain limited by the available precision. The increase of discretiza-

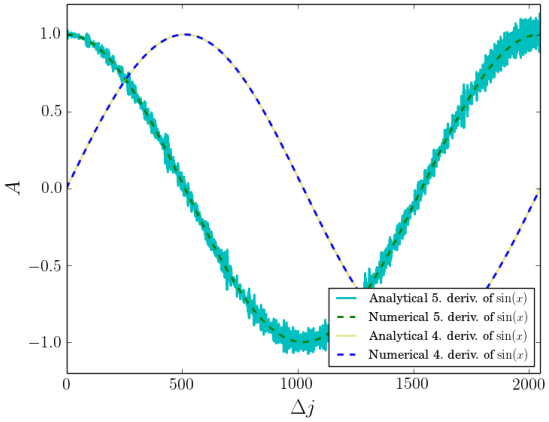


(a) Derivatives of sinusoid

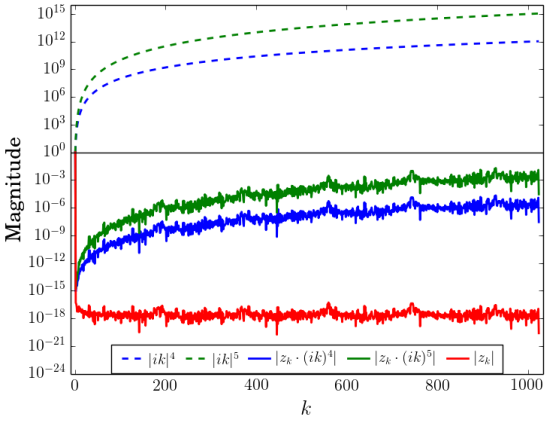


(b) FFT spectrum and wavenumber vector

Figure 1: The second derivative in single precision breaking down

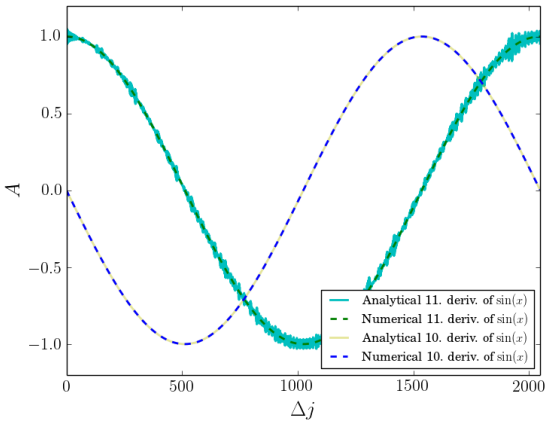


(a) Derivatives of sinusoid

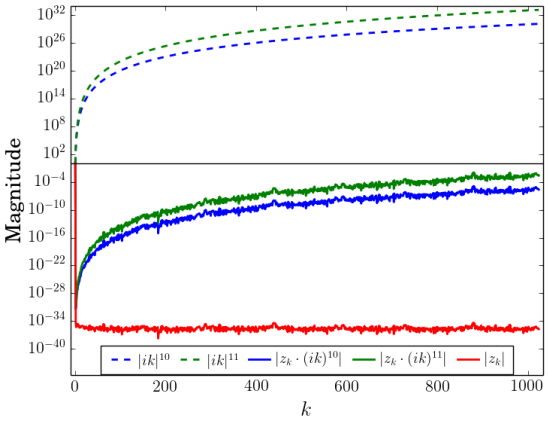


(b) FFT spectrum and wavenumber vector

Figure 2: The fifth derivative in double precision breaking down



(a) Derivatives of sinusoid



(b) FFT spectrum and wavenumber vector

Figure 3: The 11-th derivative in quadruple precision breaking down

tion can be acquired by interpolation in post-processing. The increase of information about higher wavenumbers can be obtained by only increase of precision with increase of discretization N in simulation. The tests conducted in this work indicate an expected slowdown of about one order of magnitude when using 128-bit quadruple precision instead of 64-bit double precision.

4 Conclusions

This work gives an upper possible limit of the discretization and derivatives depending on the available precision when using the Pseudospectral Fourier Method. The limit where any scheme will definitely be stable depends on the spectral content of the wave pulse transformed by FFT, as the Fourier Pseudospectral Method will amplify any numerical oscillations in the high wavenumbers. These amplified oscillations could, instead of resulting from FFT, come from the timestep scheme or appear from the solved formula (for example, the non-linear formulae will naturally introduce energy into high-wavenumber modes due to the wave steepening).

The remaining precision of the Pseudospectral Fourier Method depends on the discretization N and could also be useful in determining the precision to which the timestep scheme should converge. Alternatively if the required precision is given, it could reveal the upper limit on the spectrum which can be differentiated while retaining the minimum required precision. If more precision is required from timestep than could be given by Pseudospectral Fourier Method, the time scheme would spend unnecessary time in trying to make the result converge. Alternatively, this knowledge could be used in selecting the most economical precision.

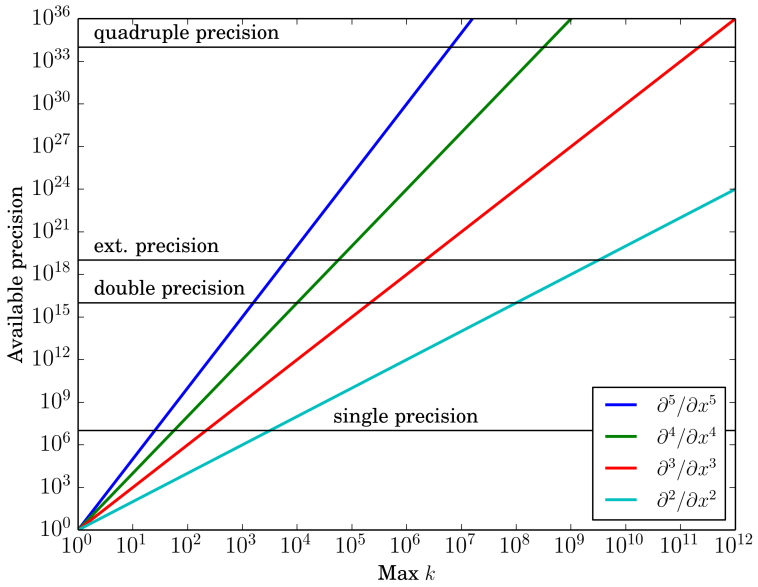


Figure 4: Maximum wavenumber k for derivatives, before the computer runs out of the precision. The actual precision of FFT routines are 2-3 magnitudes of order lower than the machine precision used.

References

[1] Andrus Salupere. The Pseudospectral Method and Discrete Spectral Analysis. In E. Quak and T. Soomere, editors, *Applied Wave Mathematics*, pages 301–333. Springer-Verlag, 2009.

[2] B. Fornberg. *A Numerical and Theoretical Study of Certain Nonlinear Wave Phenomena*. 1977.

[3] J. P. Boyd. *Chebyshev and Fourier Spectral Methods, Second Edition*. Dover Publications, Inc., 2000.

[4] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[5] Germund Dahlquist and Åke Björk. *Numerical Methods in Scientific Computing*, volume 1. Society for Industrial and Applied Mathematics, 2008.

[6] M. Frigo and S. G. Johnson. The Design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, 2005.

[7] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[8] M. Frigo and S. G. Johnson. Fft accuracy benchmark results, 2006 (accessed 17th of December 2014).

[9] D. Gottlieb and J. S. Hesthaven. Spectral methods for hyperbolic problems. *Journal of Computational and Applied Mathematics*, 128:83–131, 2001.

A Practical Guide to Solving 1D Hyperbolic Problem with Finite Element Method

Dmitri Kartofelev, Päivo Simson

1 Introduction

The following tutorial is designed to guide the student through the numerical solving of a 1D hyperbolic partial differential equation (PDE) using the finite element method (FEM). The tutorial does not provide rigorous mathematical explanations or proofs behind some aspects of FEM.

In this tutorial an initial value problem (IVP) of the wave equation is solved. The FEM approximation is realized in Python programming language. In addition the same problem is solved with FEniCS project software. The FEniCS project software that is widely used is designed to simplify and automate the solution of mathematical models based on differential equations and PDEs. After the student has introduced him or herself to this tutorial, the student should be able to solve independently other *simpler* problems, such as parabolic or elliptic problems.

2 Initial value problem

Initial value problem of the wave equation is in the form

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \tag{1}$$

$$u(x,0) = g(x), \quad \frac{\partial}{\partial x} u(x,0) = h(x) = 0, \quad \frac{\partial}{\partial t} u(x,0) = w(x) = 0 \tag{2}$$

where functions $g(x)$, $h(x)$ and $w(x)$ are the initial values distributed along the x-axis at time moment $t = 0$. We assume that space and time domains extend respectively $0 < x < L$ and $0 \leqslant t < T$. Additionally, the boundaries are considered to be fixed i.e. $u(0,t) = u(L,t) = 0$.

3 FEM

Lets solve the IVP (1), (2) numerically using FEM. The following are step-by-step instructions:

1. Construct a variational or *weak formulation*, by multiplying both sides of the PDE (1) by a test function $v(x)$ satisfying the boundary conditions (BC) $v(0) = 0$, $v(L) = 0$ to get

$$\left(\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} \right) v = 0, \tag{3}$$

and then integrating from 0 to L

$$\int_0^L \left[\left(\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} \right) v \right] dx = 0 \Rightarrow \int_0^L \frac{\partial^2 u}{\partial t^2} v \, dx - c^2 \int_0^L \frac{\partial^2 u}{\partial x^2} v \, dx = 0. \tag{4}$$

The first part of the resulting equation is left as it is, since it depends on time. The second part of the equation can be integrated by parts

$$-c^2 \int_0^L \frac{\partial^2 u}{\partial x^2} v \, dx = -c^2 \left(\frac{\partial u}{\partial x} v \Big|_0^L - \int_0^L \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} \, dx \right) = c^2 \int_0^L \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} \, dx. \tag{5}$$

From here (4) takes the following form

$$\int_0^L \frac{\partial^2 u}{\partial t^2} v \, dx + c^2 \int_0^L \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} \, dx = 0 \Rightarrow \int_0^L \left(\frac{\partial^2 u}{\partial t^2} v + c^2 \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} \right) dx = 0. \tag{6}$$

2. Generate mesh, e.g., a uniform Cartesian mesh $x_i = i\Delta x$, $i = 0, 1, \ldots, n$, where $\Delta x = L/n$, defining the intervals $[x_{i-1}, x_i]$, $i = 1, 2, \ldots, n$.

3. Construct a set of basis functions based on the mesh, such as the piecewise linear functions ($i = 1, 2, \ldots, n-1$)

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{\Delta x} & \text{if } x_{i-1} \leqslant x \leqslant x_i, \\ \dfrac{x_{i+1} - x}{\Delta x} & \text{if } x_i \leqslant x \leqslant x_{i+1}, \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

often called the hat functions.

4. Represent the approximate (FEM) solution by the linear combination of basis functions. For every fixed value of time t it must hold

$$u(x,t) \approx u_h(x,t) = \sum_{i=1}^{n-1} U_i \phi_i(x), \tag{8}$$

where the coefficients U_i are the unknowns to be determined. On assuming the hat basis functions, obviously $u_h(x,t)$ is also a piecewise linear function, although this is not usually the case for the true solution $u(x,t)$. We then derive a linear system of equations for the coefficients by substituting the approximate solution $u_h(x,t)$ for the exact solution $u(x,t)$ in the weak form (6)

$$\int_0^L \left(\sum_{i=1}^{n-1} \frac{d^2 U_i}{dt^2} \phi_i v + c^2 \sum_{i=1}^{n-1} U_i \frac{\partial \phi_i}{\partial x} \frac{\partial v}{\partial x} \right) dx = 0, \tag{9}$$

$$\sum_{i=1}^{n-1} \int_0^L \left(\frac{d^2 U_i}{dt^2} \phi_i v + c^2 U_i \frac{\partial \phi_i}{\partial x} \frac{\partial v}{\partial x} \right) dx = 0. \tag{10}$$

Now the test function $v(x)$ is chosen to be $\phi_1, \phi_2, \ldots, \phi_{n-1}$ successively, to get the system of $n-1$ linear equations:

$$\sum_{i=1}^{n-1} \int_0^L \left(\frac{d^2 U_i}{dt^2} \phi_i \phi_j + c^2 U_i \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} \right) dx = 0. \tag{11}$$

It is obvious that quantities

$$A = \int_0^L \phi_i \phi_j \, dx, \tag{12}$$

$$B = \int_0^L \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} \, dx, \tag{13}$$

can be understood and represented as matrices. System of Eqs. (11) can be written in a matrix form as follows:

$$A \frac{d^2 \mathbf{U}}{dt^2} + c^2 B \mathbf{U} = 0, \tag{14}$$

where $\mathbf{U} = (U_1, U_2, \ldots, U_{n-1})^T$ is a vector that contains coefficients U_i that are to be determined. The values of sparse matrices A and B can be obtained by taking the integrals shown

in (12) and (13). In this particular case

$$A = \begin{pmatrix} 4 & 1 & 0 & \cdots & 0 & 0 \\ 1 & 4 & 1 & \cdots & 0 & 0 \\ 0 & 1 & 4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 4 & 1 \\ 0 & 0 & 0 & \cdots & 1 & 4 \end{pmatrix}, \tag{15}$$

$$B = \frac{6}{\Delta x^2} \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}. \tag{16}$$

5. Integrate the obtained Eq. (14) with respect to time using standard ordinary differential equation (ODE) numerical integrators. The matrix equation (14) can be solved for $\mathbf{U}$, and from there one can obtain the approximate solution $u_h(x,t) \approx u(x,t)$.

   More suitable form of Eq. (14) for numerical integration is

$$\frac{d^2\mathbf{U}}{dt^2} + c^2 A^{-1} B\mathbf{U} = 0, \tag{17}$$

   where A^{-1} denotes the inverse matrix of A. The final form of FEM approximation of the wave equation (1) is thus in the following form

$$\frac{d^2\mathbf{U}}{dt^2} + C\mathbf{U} = 0, \tag{18}$$

   where $C = c^2 A^{-1} B$. This equation can be rewritten as a system of two first order equations with respect to time

$$\begin{cases} \dfrac{d\mathbf{U}}{dt} = \mathbf{V}, \\ \dfrac{d\mathbf{V}}{dt} = -C\mathbf{U}. \end{cases} \tag{19}$$

   This system of equations can in turn be solved with ODE solvers.

6. Carry out the error analysis and triple-check Your work. Congratulations You're done!

4 Implementation in Python

The following is a Python program code which is based on the results presented in the previous section. The time dependent part of the problem is solved using `scipy.integrate` package.

```
1  from scipy.integrate import *  #used for ODE integration
2  from matplotlib.pylab import * #array and matrix manipuation routines
3
4  c = 1          #Parameter in the wave equation (wave speed)
5  n = 100        #Number of mesh points (nodes)
6  L = 10.        #Space domain length
7  dx = L/(n-1) #Mesh stepsize
8  t_0 = 0.0      #Starting time (for ODE integrator)
9  t_1 = 5.0      #End time (for the ODE integrator)
10 dt = 0.05      #Time step (for the ODE integrator)
11
```

```
#Initial condition
u_0 = exp(-(2.0*(arange(0, L, dx)-L/2))**2.0) #initial value g(x)
v_0 = zeros(n-1)                               #initial value h(x)

#FEM matrix A
A = 4*eye(n-1, k=0)+eye(n-1, k=-1)+eye(n-1, k=1)

#FEM matrix B
B = 2*eye(n-1, k=0)-eye(n-1, k=-1)-eye(n-1, k=1)
d = 6*c*c/(dx*dx)
B *= d

#FEM matrix C
invA = inv(A) #A inverse matrix
C = dot(invA, B)

#Integrating the time dependent system of linear equations
def eqsys(t, u_v):
        u = u_v[:n-1]
        v = u_v[n-1:]
        u_t = v           #FEM, see Eq. (19)
        v_t = -dot(C, u) #FEM, see Eq. (19)
        u_v_t = hstack([u_t, v_t]) #reshaping vectors into a single vector
        return u_v_t
r = ode(eqsys).set_integrator('dopri5', rtol=1e-15, atol=1e-13, nsteps=10000)
u_v_0 = hstack([u_0, v_0]) #reshaping the initial value vectors
r.set_initial_value(u_v_0)
while r.successful() and r.t < t_l:
        r.integrate(r.t+dt)
        u = r.y[:n-1]  #the solution vector at time moment r.t
exit()
```

Figure 1 shows the numerical solution obtained by the aforementioned Python code. One can clearly see that the solution becomes more stable and less dispersive as the number of mesh point n grows ($\Delta x \to 0$).

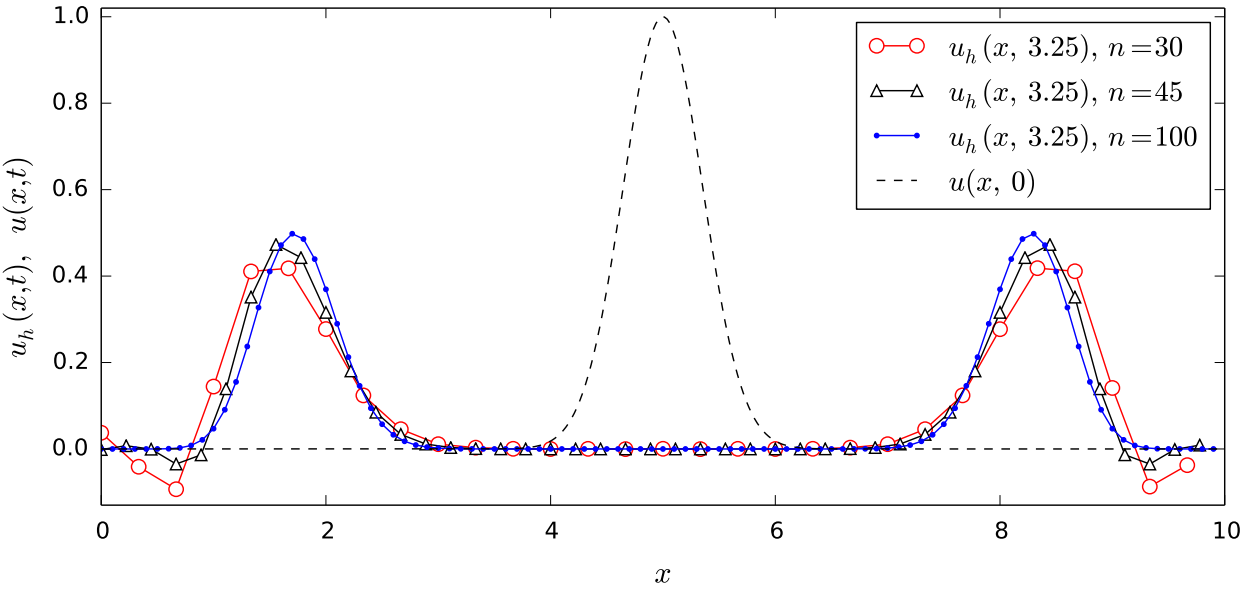


Figure 1: Numerical solution of the wave equation for bell-shaped initial value function (dashed line). Solution shown for $t = 3.25$ and for three different numbers of mesh points n: $n = 30$ (hollow bullets), $n = 45$ (hollow triangles), $n = 100$ (small filled bullets).

5 Implementation in Python using FEniCS

Using FEniCS for solving the same IVP drastically shortens the time spent on the preparation for the actual numerical calculations. If one uses FEniCS only two steps are required from the coder compared to the six steps in the previous approach that was discussed in Section 3.

Lets solve IVP (1), (2) numerically using FEniCS software package. The following are step-by-step instructions which are followed by an example Python program code:

1. Take care of the integration over time. In the case of hyperbolic (and parabolic) problem some attention needs to be allocated to the time dependent part of the equation. One can use ODE integrators in a similar manner as was shown in the previous approach. In this tutorial a simple finite difference method (FDM) is used in order to preserve clarity.

   The left-side part of wave equation (1) needs to be approximated by two steps backwards finite differences

   $$\frac{\partial^2 u}{\partial t^2} \approx \frac{u^k - 2u^{k-1} + u^{k-2}}{\Delta t^2}, \tag{20}$$

   here $u^k = u(x, k\Delta t)$, $u^{k-1} = u(x, (k-1)\Delta t)$ and $u^{k-2} = u(x, (k-2)\Delta t)$, where the index $k = 1, 2, \ldots, m$ and $\Delta t = T/m$.

   Using (20) the wave equation is rewritten in the following form

   $$\frac{u^k - 2u^{k-1} + u^{k-2}}{\Delta t^2} = c^2 \frac{\partial^2 u^k}{\partial x^2} \Rightarrow u^k - 2u^{k-1} + u^{k-2} = c^2 \Delta t^2 \frac{\partial^2 u^k}{\partial x^2} \Rightarrow \tag{21}$$

   $$\Rightarrow u^k + c^2 \Delta t^2 \frac{\partial^2 u^k}{\partial x^2} = 2u^{k-1} - u^{k-2}. \tag{22}$$

2. Construct a variational or *weak formulation* of the Eq. (22). See Step 1 in Section 3.

   $$\int_0^L u^k v \, dx + c^2 \Delta t^2 \int_0^L \frac{\partial u^k}{\partial x} \frac{\partial v}{\partial x} \, dx = \int_0^L \left(2u^{k-1} - u^{k-2}\right) v \, dx, \tag{23}$$

   where v was the test function. Obtained mathematical expression (23) of the variational formulation is directly inserted into the program code and the FEniCS solves it by taking care of everything else that is required.

3. Congratulations You're done!

```
1  from dolfin import *   #Dolphin interface imports FEniCS
2  import numpy as np     #NumericPython
3
4  c = 1       #Parameter in the wave equation (wave speed)
5  a = 0.0     #Space variable
6  L = 10.0    #Space variable
7  n = 1000    #Space variable number of mesh points
8  dt = 0.01   #Time stepsize
9  t = 0.0     #Time variable
10 T = 5.0     #Time variable
11
12 #Initial condition
13 x = np.linspace(a, L, num = n + 1)
14 u0 = np.exp(-4*(x - 5)*(x - 5))
15
16 #Mesh and function spaces
17 mesh = IntervalMesh(n, a, L)
18 V = FunctionSpace(mesh, 'CG', 1)  #FEniCS generates all necessary for the mesh
19
20 #Previous solutions for time integration
21 uk1 = Function(V)
```

```
22 uk2 = Function(V)
23 uk1.vector()[:] = u0
24 uk2.vector()[:] = u0  #Initial velocity is 0
25
26 uk = Function(V)       #Current solution
27
28 #Variational problem at each time moment (FEM)
29 u = TrialFunction(V)
30 v = TestFunction(V)
31 const = c*c*dt*dt
32 A = u*v*dx + const*inner(grad(u), grad(v))*dx  #Weak formulation, see Eq. (23)
33 L = (2.*uk1 - uk2)*v*dx                        #Weak formulation, see Eq. (23)
34
35 #Time integration and FEniCS solver
36 while t <= T:
37         solve(A == L, uk)  #FEniCS solver
38         uk2.assign(uk1)    #Needed for time integration, u^(k-2)
39         uk1.assign(uk)     #Needed for time integration, u^(k-1)
40         t += dt
41         plot(uk)           #FEniCS plots the FEM solution at time t
```

Additional useful feature of the FEniCS is that it can solve 2D, 3D or higher dimensional problems. For more detail refer to the FEniCS manual. This feature is especially useful in various engineering applications.

6 Conclusions

A short tutorial for solving the 1D wave equation numerically was presented. The IVP (1), (2) was successfully solved using FEM.