SC19-20

Leszek Marcinkowski

May 29, 2020

Contents

1 Introduction

These notes are for Scientific Computing course in University of Warsaw in 2020. Due to the extraordinary situation there no real meetings.

In those notes I will briefly write what every participant should read.

2 Numerical Linear Algebra reminder and what can be done in octave

1. Solving linear systems backslash or linsolve ()
2. How is an inverse of nonsingular matrix computed and octave's function inv(A) -

March 10

3. matrix factorization LU, L^TL (Cholesky), QR and what are they used for. Octave: lu (), qr (), chol () functions - March 10
4. LLSP (Linear Least Square Problem) a reminder, when there is a unique solution, what the backslash operator return in case the matrix is not of maximal rank etc - March 10.
5. SVD (Singular Value Decomposition) what's this, what is ti used for etc - see below a short description - can be read in these notes - March 24.
6. Sparse matrices formats in octave and in general - in these notes March 24.

2.1 SVD and LLSP

LLSP is to find u^* s.t.

$$u^* = arg \min_u \|Au - f\|_2 \tag{1}$$

for a given A $m \times n$ and $f \in R^m$. If the null space $N(A) = \{0\}$ then there is a unique solution, otherwise we have a set a solutions which is $u + N(A)$ with u being any solution, then usually to make a problem unique we want to get the solution of minimal second norm. The main octave's solver is the backslash operator: u=A\b.

In case of $N(A)$ being a zero subspace the main way of solving LLSP is to get the QR factorization of A and then there is a direct formula for the solution. Otherwise we can apply SVD decomposition of A. What is it? Namely SVD are 2 orthogonal matrices U, V and a diagonal matrix D such that

$$A = UDV^T,$$

the nonzero elements of D (assume that they are the first r elements of the diagonal of D : $\sigma_1 \geq, \ldots \geq \sigma_r > 0$ are strictly positive and are called the singular values of A. They are unique. The orthogonal matrices U, V may or may not be unique depending on the matrix A.

Then using SVD (what can be obtained in octave using the function svd(A)) we can compute

- the 2nd norm of A: namely $\|A\|_2 = \sigma_1$; in octave norm(A) or norm(A,2).
- if A $n \times n$ is quadratic nonsingular then we can get the cond number of A: $cond_2(A) = \sigma_1/\sigma_n$ (here all singular values are positive); in octave cond(A)

- if A $n \times n$ is quadratic nonsingular then we can get the inverse of $A^{-1} = VD^{-1}U^T$ (here all singular values are positive so D is nonsingular); in octave inv(A) but it is computed by a much cheaper algorithm...
- the rank of A - r the no of strictly positive singular values (naturally it is really a numerical rank, we have to define a threshold such that the computed singular values which are below it are considered as zero; in octave rank(A)
- the orhonormal basis of $R(A)$- the space spanned by the columns of A - the first r column of U; in octave orth(A)
- the orhonormal basis of $N(A)$ - the null space of A - the last $n - r$ columns of V; in octave null(A)
- the solution of LLSP with A $m \times n$ and a rhs vector $\vec{b}$ of minimal second norm, namely:
$$u^* = V * (g_1/\sigma_1, \ldots, g_r/\sigma_r, 0, \ldots, 0)^T \qquad \vec{g} = U^T * \vec{b}$$
is the unique solution of the minimal second norm of this LLSP; in octave A\b.
All solutions of LLSP are given as
$$\{V * (g_1/\sigma_1, \ldots, g_r/\sigma_r, u_{r+1}, \ldots, u_m)^T : u_{r+1}, \ldots, u_n \in R\} \qquad \vec{g} = U^T * \vec{b}.$$
In octave A\b+null(A)*v where v any vector of $n - r$ dimension.

We leave as exercises to prove those assertions.

2.2 Sparse formats of matrices

In many applications there appear naturally matrices with a lot of zeros, e.g. in PDEs discretizations, thus e.g. solving the linear system with such a matrix using standard solvers like LU would require unnecessary memory usage and computational effort. Thus there are special formats ways of storing only non-zeros of a matrix.

In general there are several sparse formats of matrices i.e. special ways of storing matrices which have a lot of zeros elements. In octave there is possibility of storing a matrix in a sparse format. In order to convert a standard matrix into a sparse one (actually it stays the same matrix just the storage is changed) is octave's function: sparse(). In order to convert a sparse matrix to a standard one (again just the way the elements of the matrix are stored is changed) is the function full (). Some functions automatically use special usually cheaper algorithms when one of its argument is in the sparse format (in octave there is one format, an user does not have to know it) e.g. backslash. There are

several functions specially designed for sparse formats, e.g. spdiags() allows creating a sparse banded matrices, spy() plots the pattern of the sparsity etc. I recommend reading a chapter in the octave documentation on sparse formats.

The 4 most popular formats of sparse matrices are:

- diagonal format: only for banded matrices - the diagonals of a banded matrix are kept in vectors e.g. tridiagonal matrix can be kept as 3 vectors with its diagonals (one can claim it no really a sparse format)
- coordinate format (COO) or triple format or ijv format: nonzero elements of a matrix A are kept in three vectors of NNZ (no of nonzeros) dimension : two integer vectors: I, J of dimension NNZ and a real vector V of dimension NNZ in which there are nonzero elements of the matrix - then we have the following

  $$A(I[k], J[k]) = V[k] \qquad k = 1, \ldots, NNZ,$$

  i.e. the integer vectors have on the kth position the numbers of the row and column respectively of the kth element of V. This format is not used by octave but surprisingly in a main creating sparse matrix function of octave it can be used for creating a matrix i.e. an user have to prepare 3 vectors I, J, V as in COO and then A=sparse(V,I,J,m,n) (m, n are optional - the dimension of the matrix A if not provided $m = max_k(I[k]), n = max_k(J[k])$) creates a sparse matrix but in another interior octave's format.
- the compressed sparse row format (CSR): we have also 3 vectors for storing a matrix A $m \times n$ of NNZ nonzero elements. A real vector V and integer vector J of dimension NNZ. In V we have nonzero elements kept in packed rows and in J indices of columns of the respective elements in V , i.e. an element $V[k]$ is in the $J[k]$ column of A. In the 3rd integer vector IP of the dimension $m + 1$ we have the no of indices of V, J pointing to the first elements of respective rows. Usually, the first entry of IP equals zero (assuming that vectors V, J are numbered $0, \ldots, NNZ-1$) and the last one NNZ (this is an additional auxiliary entry), then $IP[i+1] - IP[i]$ $i = 0, \ldots, m-1$ equal to the no of non-zeros of the (i+1)-th row. The (i+1)-th row nonzero elements are $V[IP[i]]$ to $V[IP[i+1]-1]$, their columns indices equal $J[IP[i]]$ to $J[IP[i+1]-1]$. Thus if $IP[i] \leq k < IP[i+1]$ then

  $$A(i+1, J[k]) = V[k].$$

- compressed sparse column (CSC) format is analogous to CSR one - the place of the rows is taken by the columns - the nonzero values are kept in packed columns, we have an integer vector keeping the no of rows and an integer vector in which we have numbers pointing to the first elements of respective columns in the other two vectors. This is the format used by octave.

There are other formats but we will not discuss them.

3 Nonlinear solvers in octave

In octave the main two nonlinear solvers are the function fzero () and fsolve ().

fzero () is the main solver for solving one equation, i.e finding a zero of univariate function:

$$f(x) = 0$$

We must provide a function (in a form of a function handle to a function or the string with its name) a two dimensional vector $x0 = [x1; x2]$ such that f changes sign on the interval $x1 < x2$, if an user gives $x0$ a scalar the solver tries to find another point $x2$ such that the signs of f in $x0$ and $x2$ are different. The function returns an approximation of the zero and optionally return code informing if everything is OK etc

fsolve () is a solver of any system of equations. A user must provide the function which for a given vector x returns $f(x)$ and an initial approximation $x0$. It returns an approximation of a root and optionally an info etc

There also some minimizers in octave e.g. fminbnd(), fminunc().

I refer to the octave documentation for details.

3.1 Inverse function

It happens that we want to compute a value of an inverse of a given function $f : [a, b] \subset R \to R$ for $y \in [f(a), f(b)]$, e.g. it is a usual case if we want to solve ODEs by the separation of variables method. Usually we cannot explicitly represent a formula for f^{-1}. Then in order to compute $f^{-1}(y)$ we can try to solve the following equation (or a system of equations): find x such that

$$f(x) = y \qquad or \qquad f(x) - y = 0$$

Naturally x solving such equation equals $f^{-1}(y)$. We have the same situation of we want to compute a value of the inverse of multidimensional function $F : D \subset R^m \to R^m$ - then we have to solve the system of equations

$$F(X) - Y = 0$$

then $X = F^{-1}(Y)$.

If we want to plot a graph of the inverse of an univariate function f then it is enough to plot the points $(f(x), x)$ for sufficiently dense points in x, but if the function is very steep then the graph would not look nice. More details are in chapter 3 of the lecture notes [**?**].

In octave we have to use appropriate nonlinear solvers.

3.2 Implicit function

Sometimes we want to compute a value of implicitly given function: we want to compute the value of f at x for an implicit function f given by

$$F(x, f(x)) = 0$$

where $F : R^n \times R^m \to R^m$. The implicit function is unique if we have e.g. an additional condition, e.g. we know one of its values $F(x_0, y_0 = f(x_0) = 0$ plus some conditions on the regularity and derivatives of F at (x_0, y_0). A surface in R^3 or a curve in R^d $d = 2, 3$ may be defined that way, e.g. $F(x, y) = x^2 + y^2 = 0$ with $F(0, 1) = 0$ defines a curves $y(x) = \sqrt{1 - x^2}$ on $(-1, 1)$. Thus for a given x we have to solve the system of equations for y:

$$g(y) := F(x, y) = 0$$

and $y = f(x)$.

In octave we have to use appropriate nonlinear solvers.

3.3 A short introduction to the lab problem

We want to solve the problem described in chapter 3 of the lecture notes [**?**]. Namely, we want to plot a graph of the inverse function to

$$\Sigma(s) = \left(\frac{2}{M-1}\right) * (1 - (1-s)^{M-1}) - s * (1-s)^{M-1}$$

for $M = 0.5$ on $[0, 60]$. It can be done by plotting $(\Sigma(s), s)$ or by solving the respective scalar equation $\Sigma(s) - z = 0$ for $s = \Sigma^{-1}(z)$. But specially for large z it is expensive,

thus one can note that our scalar equation is equivalent to another one namely

$$G_z(s) = 1 - s - \left(\frac{p-s}{p+z}\right)^{p/2} \qquad p = 2/(1-M)$$

which is easier to solve for standard solvers. The last way is to use a hammer, i.e. generate a lot of points clustered near one and use the first method.

4 Solving ODEs or IVPs in octave

4.1 IVP in octave

The IVp is to find $x(t)$ solving ODE: $x' = f(t,x)$ with an initial condition $x(t_0) = x_0$. THe right hand side function fro a given time t and vector x returns a vector. The main tool for solving Initial Value Problem in octave is the function lsode(). The simplest call is X=lsode(f,x0,t) where f is a function handle to the vector field function, i.e. function y=f(x,y) which for a given vector x and time t returns the vector of the same length as x. $x0$ is the vector from the righ hand side of the initial condition, and $t = (t_0, t_1, \ldots, t_n)^T$ is a vector of discrete times, note that the first entry is equal to t_0 from the initial condition. The function returns X a matrix $n \times m$ with the approximations of the solution at discrete times rom the vector t. The kth row contains an aproximation of $x(t_k)^T$.

x=lsode(@(x,t) x ,1,[0,1])

This call to lsode() computes the approximation of the solution of $x' = x, x(0) = 1$ at $t = 1.$, the function returns two values $(x_0, x_1)^T$ with $x_0 = 1, x_1 \approx x(1) = \exp(1))$,

If we want to compute numerical slution at more discrete times we have to give more values at the vector t:

x=lsode(@(x,t) x,1,t=0:0.1:1)

Then we would get the approxiamtion of the solution at all $t_k = k*0.1$. It is important to note that the exactness of the method is independent of the number of time points, it is set in

lsode_options()

function with some default tolerances values.

If we want to solve a hih=gher order ODE then we have to convert the ODE t othe first order ODE by adding extra variables, e.g. let consider $x'' = -x, x(0) = 0, x'(0) = 1$ then it is equivalent to the following first order ODE:

$$\begin{aligned} x' &= y \\ y' &= -x \\ x(0) &= 0 \\ y(0) &= 1 \end{aligned}$$

This is a linear system $X' = AX$ with

$$A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Then we can solve this system as folllows:

X=lsode(@(x,t) [0,1;−1,0]∗x ,[0;1],[0, pi]) .

This would return matrix 2×2 with an approximation of $(x(1), x'(1))^T$ in the second row, in the first row we have the transpose of the initial condition.

4.2 Earthquake model

4.2.1 A model of earthquake

We have building with n floors. During an earthquake acts a force $G\sin(\gamma t)$ on the 1st floor. Between jth and $j+1$th floor acts a restoration force f_j proportional to relative shift of the floors:

$$f_j = k_{j+1}(x_{j+1} - x_j)$$

f_j acts on jth floor - $-f_j$ on the $j+1$th. On the first floor acts analogous force $-f_0 = -k_1 x_1$ between ground and this floor. Thus on each floor we get

$$m_j x_j = f_j - f_{j-1} = k_{j+1}(x_{j+1} - x_j) - k_j(x_j - x_{j-1}) = k_j x_{j-1} + (-k_{j+1} - k_j)x_j + k_{j+1}x_{j+1}$$

with $k_{n+1} = 0$ and $x_0 = 0$ (we assume that earth does not move or rather that floor moves vs ground).

All together we get the system :

$$Mx'' = Sx$$

with $M = diag(m_1, \ldots, m_n)$ a diagonal matrix with the masses of the floors on the diagonals and S the tridiagonal matrix with $(-k_{j+1} - k_j)_{j=1,\ldots,n}$ and the main diagonal ($k_{n+1} = 0$), $(k_j)_{j=2,\ldots,n}$ on subdiagonal and $(k_j)_{j=1,\ldots,n-1}$ on the superdiagonal:

$$S = \begin{pmatrix} -k_1 - k_2 & k_2 & & & \\ k_2 & -k_2 - k_3 & k_3 & & \\ & \ddots & \ddots & \ddots & \\ & & k_{n-1} & -k_{n-1} - k_n & k_n \\ & & & k_n & -k_n \end{pmatrix}$$

In case of an earthquake the system looks as:

$$Mx'' = Sx + F(t)$$

with

$$F = \begin{pmatrix} G\sin(\gamma t) \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

γ is the frequancy of the earthquake, e.g. equal to 3 - and the period of the earthquake $T = 2\pi/\gamma$ usually in the range $[2, 3]$, G its magnitude (in practise F should quickly disappear but we consider a short time during the earthquake to see if the bldg collapse or not).

We can multiply the system by M^{-1} and get

$$x'' = M^{-1}Sx = Ax \quad x'' = Ax + M^{-1}F$$

We can compute the eigenvalues of A : λ_j and their square roots

$$\omega_i = \sqrt{-\lambda_j}$$

which we call the frequencies of the building and

$$T_i = 2\pi/\omega_j$$

the periods of the building. It is important that the periods are NOT in the range of the earthquake period which is usually in $[2, 3]$.

4.2.2 Resounance in 1th storey building

For simplicity consider the case of one floor building with e.g. $m = 10000kg$ and $k = 5000kg * m/s^2$, thus our equation $x'' = (-k/m)x + (G/m)sin(\gamma t)$ the general solution

(why?) is

$$c_1 \sin(at) + c_2 \cos(at) + x_0(t) \quad (a = k/m)$$

where the sum of the 1st two =terms is called the general solution of the homogeneous problem (rhs =0) and x_0 is any specific solution of the equation - because the rhs is of the form G/m can be found in the following form:

$$x(t) = d_1 \sin(\gamma t) + d_2 \cos(\gamma t)$$

if $\gamma \neq \sqrt{a}$! Substituting x_0 into the ODE we get:

$$x_0'' + ax_0 = d_1(-\gamma^2 + a)\sin(\gamma t) + d_2(-\gamma^2 + a)\cos(\gamma t) = (G/m)\sin(\gamma t)$$

We get $d_2 = 0$ and $d_1 = (G/m)/(-\gamma^2 + a)$. - we see that our assumption of $\gamma \neq \sqrt{a}$. The constants c_k can be computed from the initial values. Thus we get bounded oscillatory solutions.

In case $\gamma = \sqrt{a}$ - we know (ODE course)

$$x_0(t) = (d_1 + d_2 t)\sin(\gamma t) + (d_3 + d_4 t)\cos(\gamma t)$$

Again after substitution we get the linear system for d_k. Note that at least one of d_2 or d_4 have to be nonzero (why?). Thus we get that t_0 is unbounded...this physical phenomenon is called resonance...

In general for many storey building we obtain similar (much more complicated) formulas and in case when the earthquake frequency equals (mathematically - in real life is close) to one of frequency of the building we can resonance and the building may collapse.

4.3 Some info on schemes for ODEs or IVPs

We want to solve IVP:

$$x' = f(t, x) \quad x(t_0) = x0$$

We introduce discrete times: $t_k^h = t_k = t_0 + k * h$ for positive step h and we want to approximate $x(t_k)$ by $x_k^h = x_k$ (we usually omit the superscript h if h is defined) computed by some scheme.

4.3.1 Some basics

If we replace the derivative by the forward difference:

$$\frac{x(t+h)-x(t)}{h} \approx x'(t) = f(t, x(t))$$

for $t = t_k$ then we obtain the simplest scheme: which is the Euler one (or explicit Euler scheme):

$$\frac{x_{k+1}-x_k}{h} = f(t_k, x_k)$$

or

$$x_{k+1} = x_k + hf_k$$

where $f_k = f_k^h = f(t_k, x_k)$. Knowing $x_0^h = x_0$ from the initial condition we can compute the succesive $x_k, k \geq 1$ as long as f_k is properly defined (we can get outside the domain of f) or get blow up.

The other way to obtain the Euler scheme (called also forward Euler or explicit Euler scheme) is to use the Taylor formula:

$$x(t+h) = x(t) + x'(t)h + O(h^2) = x(t) + f(t, x(t))h + O(h^2)$$

forgetting about the remainder we get:

$$x(t_{k+1}) \approx x(t_k) + hf(t_k, x(t_k))$$

replacing $x(t_k)$ with x_k we get our scheme.

One can ask why to to use forward difference - we can replace the derivative with a backward difference:

$$x'(t) \approx \frac{x(t)-x(t-h)}{h}$$

then we get the following scheme

$$\frac{x_k - x_{k-1}}{h} = f_k$$

or

$$x_{k+1} = x_k + hf_{k+1}$$

- the implicit (backward) Euler scheme. Implicit means that we have no a direct (explicit) fomula for x_{k+1}. There may be no solution for a given x_k in general and it must be computed by solving the following nonlinear system of equations:

$$g(x) := x - x_k - hf(t_k, x) = 0$$

Fortunately, under reasonable assumptions there is a solution for sufficiently small h and the system is getting simpler to solve as h is getting smaller.

One might ask why at all consider such a scheme if we have explicit schemes? It is not obvious but there is large class of IVPs such that implicit schemes work much better for them than explicit ones.. They are called stiff equations.

Let's go back to the Taylor expansion :

$$x(t+h) = x(t) + x'(t)h + 0.5x''(t)h^2 + ..$$

to get the explicit Euler scheme we took two terms but we could try to take three or more, e.g.

$$x(t+h) \approx x(t) + x'(t)h + 0.5x''(t)h^2$$

in order to construct a new scheme. The question how to compute e.g. $x''(t)$ for x the solution of ODE: $x' = f(t,x)$? We can differentiate the equation getting:

$$x''(t) = \frac{d}{dt}x'(t) = \frac{d}{dt}f(t,x(t)) = f_t(t,x(t)) + f_x(t,x(t))x'(t) = f_t(t) + f_x(t,x(t)) * f(t,x(t)).$$

Thus we can define the following scheme:

$$x_{k+1} = x_k + hf_k + 0.5h^2 * (f_t(t_k,x_k) + f_x(t_k,x_k)f_k).$$

In order to compute x_{k+1} we have to compute the value of $f_k = f(t_k,x_k)$ (evaluating all components, i.e. m univariate functions at (t_k,x_k)) $f_t(t_k,x_k)$ the same cost more or less and $f_x(t_k,x_k)$ (evaluating $m \times m$ univariate functions in general).

Let define the concept of convergence of the scheme. Our scheme generates the approximate solutions $x_k = x_k^h$ for any h which approximates $x(t_k)$ with $t_k = t_k^h = t_0 + k*h$ on $[0,T]$ for $k \leq (T-t_0)/h$. We say that the scheme is convergent if

$$E_h = \max_{t_k^h \in [0,T]} \|x_k^h - x(t_k^h)\| \to 0 \quad h \to 0$$

and the convergence is of order $p > 0$ if additionally

$$E_h = O(h^p).$$

Usually p is an integer but formally may be fractional.

Note that we do not define the norm as it can by any norm in R^m - we would like to recall that all norms are equivalent in the finite dimensional spaces, however the equivalence constants may be very arbitrarily large/small: just consider $\|x\|_1$ and $\|x\|_\infty$, then the (optimal) equivalence constants are

$$\|x\|_\infty \leq \|x\|_1 \leq m\|x\|_\infty.$$

Thus in practise we get the same convergence order in all norms, but the real errors may be many orders larger/smaller for the same h but different norms.

There is also an important property of the so called the order of the scheme (not the convergence order!). More or less we substitute the solution at any $t_n = t \in [t_0, T)$ into the scheme terms and measure the residue it as $O(h^p)$. As p larger then the error is smaller. We define the order of the Euler schemes. Thus if for sufficiently smooth x we have that

$$e_h = \max_{t\in[t_0,T-h)} \|\frac{x(t+h)-x(t)}{h} - f(t,x(t))\| = O(h^p)$$

we say that the Euler scheme is of order p, one can show that here $p = 1$. Equivalently we can demand that:

$$he_h = \max_{t\in[t_0,T-h)} \|x(t+h) - x(t) - hf(t,x(t))\| = O(h^{p+1}).$$

The implicit Euler scheme is also of the first order and the above Taylor scheme is of second order.

4.3.2 One step schemes - Runge-Kutta schemes

The major class of one step schemes i.e. the schemes given as :

$$x_k = \Phi(h, t_n, x_n, x_{n+1})$$

for some function Φ is the class of Runge-Kutta schemes.

We will present the idea of the simplest Runge's scheme.

We want t ouse an extra value of f. I.e. we compute the value of f at

$$\hat{x} = x_k + ahf(t_k, x_k)$$

which is a one step of the explicit Euler scheme with the step ah. We get $\hat{f} = f(t+ah, \hat{x})$ and then we define

$$x_{k+1} = x_k + h * f(c_1 f_k + c_2 \hat{f})$$

i.e. we add to x_k the linear combination of f_k and $\hat{f}$ in such a way to obtain a better scheme. To get a Runge scheme we select c_1, c_2 to get a scheme of the highest possible order. Here we can get only the scheme of order two - there is a whole family of explicit

Runge schemes of order two. Here we give formulas for two most popular ones: the Heun scheme: $c_1 = c_2 = 0.5$ and $a = 1$

$$x_{k+1} = x_k + 0.5h * (f_k + f(t + h, x_k + hf_k))$$

and the modified Euler scheme: $c_1 = 0, c_2 = 1$, $a = 0.5$:

$$x_{k+1} = x_k + hf(t + 0.5h, x_k + 0.5hf_k).$$

Sometimes any explicit second order Runge scheme is called a modified Euler scheme. There are also higher order Runge schemes, e.g.: here we see an example of a fourth order Runge scheme:

$$\begin{aligned} x_{k+1} &= x_k + \tfrac{h}{6} * (K_1 + 2K_2 + 2K_3 + K_4) \\ K_1 &= f(t_k, x_k) \\ K_2 &= f(t_k + 0.5h, x_k + 0.5hK_1) \\ K_3 &= f(t_k + 0.5h, x_k + 0.5hK_2) \\ K_4 &= f(t_k + h, x_k + h * K_3) \end{aligned}$$

which is a scheme of order four. There are also implicit Runge schemes.

4.3.3 Linear multistep schemes - Adams schemes

Another important schemes are so called linear multistep schemes. The main examples of this class are Adams schemes.

They are derived form the formula:

$$x(t + h) = x(t) + \int_t^{t+h} x'(s)ds = x(t) + \int_t^{t+h} f(s, x(s))ds$$

If we substitute $x(t+h)$ by x_{k+1} and $x(t)$ by x_k and $f(s, x(s))$ by a simple approximation $Q_p(s) \approx f(s, x(s))$ which can be computed knowing the values of $f(t_l, x_l)$ for $l = 0, -1, .., p$ getting an explicit scheme or $l = 1, 0, -1, ..$ getting an implicit one. In Adams scheme Q_p is a Lagrange interpolation polynomial interpolating $f_l = f(t_l, x_l)$ at the previous $p + 1$ time steps , i.e.

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} Q_p(s)ds$$

where Q_p is a polynomial of degree less or equal p such that

$$Q_p(t_l) = f_l \quad l = k, k - 1, k - 2, \ldots, k - p$$

getting an explicit Adams-Bashford schemes or

$$Q_p(t_l) = f_l \quad l = k + 1, k, k - 1, k - 2, \ldots, k - p + 1$$

getting an implicit Adams-Moulton scheme. Note that the coefficients of Q_p in $(t-x_k)^r$ basis of the polynomial space of degree less or equal p. are uniquely defined by f_l and h i.e .the values at the interpolation nodes. Thus after integrating we get:

$$x_{k+1} = x_k + h * (\beta_p f_k + \ldots + \beta_0 f_{k-p}) \qquad (Adams - Bashford)$$

or

$$x_{k+1} = x_k + h * (\hat{\beta}_p f_{k+1} + \ldots \hat{\beta}_0 f_{k-p+1}) \qquad (Adams - Moulton)$$

After renumbering we get a $p+1$ step explicit scheme:

$$x_{k+1+p} = x_{k+p} + h * (\beta_p f_{k+p} + \ldots + \beta_0 f_k) \qquad (Adams - Bashford)$$

or a p-step implicit scheme:

$$x_{k+p} = x_{k+p-1} + h * (\hat{\beta}_p f_{k+p} + \ldots + \hat{\beta}_0 f_k) \qquad (Adams - Moulton)$$

Let's compute the first two A-B schemes: let take $p = 0$ then $Q_0(s) = f_k$ and we get

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} Q_0(s)ds = x_k + h * f_k$$

the explicit Euler scheme. Take $p = 2$ we get

$$Q_1(s) = f_k * \frac{s - x_{k-1}}{h} + f_{k-1} * \frac{s - x_k}{-h}$$

after integrating we get

$$x_{k+1} = x_k + 0.5h * (3 * f_k - f_{k-1})$$

the two-step Adams-Bashford scheme of order two.

Now let consider Adams-Moulton examples, take $p = 0$ and analogously we get the implicit Euler scheme:

$$Q_0 = f_{k+1}$$

after integrating we get that

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} Q_0(s)ds = x_k + h * f_{k+1}$$

Let's take $p = 1$ then

$$Q_1 = f_k \frac{s - x_{k+1}}{-h} + f_{k+1} \frac{s - x_k}{h}$$

After integrating we get the trapezoid scheme:

$$x_{k+1} = x_k + 0.5h * (f_k + f_{k+1})$$

which is of order two.

Note that in each successive steps we have to compute ONLY ONE value of f since the previous ones were computed in previous steps.

However we need some x_1 an extra starting value. So multistep schemes are not self-starting. In practise we use a starting procedure using some one-step scheme of sufficiently high order. in order to compute starting values $x_1, \ldots, x_{q-1}$ for q-step scheme (x_0 is given in the initial condition). Surprisingly it is OK to use a one step scheme of order $k-1$ for starting multistep scheme of order k.

The linear multistep schemes are cheaper than e.g .Runge schemes but less stable and less useful for being use as a base for a construction of an adaptive step scheme. They also require extra starting procedure, what is not a serious fault but is necessary.

4.3.4 Convergence theory

For one step scheme

$$x_{k+1} = x_k + h\Phi(h, t_k, x_k, x_{k+1})$$

is convergent if is consistent i.e.

1. Φ continuous
2. $\Phi(0, t, x) = f(t, x)$
3. Φ is Lipschitz with respect to the last two variables

If additionally the scheme is of order p we can get the convergence of order p for sufficiently smooth solutions.

There is a separate convergence theory for linear multistep schemes. We introduce a concept of 0-stability of a such scheme. Namely if we consider the following linear q-step scheme:

$$\alpha_q x_{k+q} + \ldots + \alpha_0 x_k = h(\beta_q f_{k+q} + \ldots \beta_0 f_k)$$

with $\alpha_q \neq 0$ then this scheme is 0-stable (or just stable) if all the roots of the following polynomial:

$$\rho(x) = \alpha_q x^q + \ldots + \alpha_0$$

are bounded by one, moreover the roots with the absolute value one are single roots.

If the q-step linear scheme is of order $p \geq 1$, is stable, and the starting values are such that $\|x_k - x(t_k)\| = O(h^p)$ for $k = 0, \ldots, q-1$, then we get the convergence of order p is the solutions is sufficiently smooth.

4.3.5 Stiffness

What is stiffness or stiff equation? In 1950ties it was noticed that for some system of ODEs explicit schemes work very bad or do not work at all, but the implicit schemes work much better. So the practical definition of stiff systems is that they the ones for which explicit schemes do not work. Below we will give also a more mathematical one however this practical one very well explains what stiffness is.

First consider a very simple 1D IVP: $x' = ax$ with $x(0) = 1$ for $a < 0$ and apply the explicit Euler scheme:

$$x_n = x_{n-1} + ax_{n-1} = (1 + ha)^n$$

The solution of this IVP is $\exp(at)$ which is positive and $\lim_{t\to 0} x(t) = 0$ we would expect that our numerical solution preserve those properties, but to get $x_n > 0$ for all n we should have

$$h < \frac{1}{-a}$$

then also x_n tends to zero for n tending to infinity. This condition is very restrictive for a with large $|a|$ when the solution $\exp(at)$ convrge to zero very fast. If we apply the implicit Euler scheme we get that scheme:

$$\begin{aligned} x_n &= x_{n-1} + ahx_n, \\ (1-ah)x_n &= x_{n-1}, \\ x_n &= (1-ah)^{-1}x_{n-1} \\ x_n &= (1-ah)^{-n}. \end{aligned}$$

We see that $x_n > 0$ and converging to zero with n tending to infinity for any h. We have no condition on the step.x'=

Let now consider the linear system

$$x' = Ax \quad x() = x_0$$

Let assume that A is $m \times m$ diagonal matrix in a certain basis i.e.

$$A = CDC^{-1}$$

for a nonsingular matrix C. The diagonal matrix D contains eigenvalues of A on the diagonal.

Then the solution of IVP is:

$$x(t) = \sum_k \vec{c_k} \exp(\lambda_k t)\alpha_k$$

where $\vec{c_k}$ is the k-th column of the matrix C, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_m)^T = C^{-1}\vec{x}_0$.

Let assume that all those eigenvalues are negative of different order: $\lambda_1 < \ldots \leq \lambda_m < 0$. Then all the components of the solution converge to zero with $t \to 0$. If we apply the explicit Euler scheme we get:

$$x_n = \sum_k \vec{c_k}(1 + ah)^n \alpha_k$$

in order to get that the k-th discrete component converge to zero we have to take the step h:

$$h < \frac{2}{-\lambda_k} \quad k = 1, \ldots, m.$$

This the largest in the absolute value component of the solution (here it is the one for λ_1) vanishes very very quickly but force us to use very small restrictive step $h < 1/(-\lambda_1)$. If we apply the implicit Euler scheme we get:

$$x_n = \sum_k \vec{c_k}(1 - ah)^{-n} \alpha_k$$

thus we get no restriction pn the step.

Thus we can define that the system of ODEs

$$x' = Ax$$

is stiff if

- all eigenvalues of A : λ_k are negative
- the following is satisfy

$$\frac{\max_k |\lambda_k|}{\min_k |\lambda_k|} >> 1$$

  How it is large more or less measure how the system is stiff

In case of nonlinear ODEs: $x' = f(t, x)$ we can define stiffness analogously for the linearized equation i.e. for eigenvalues of Jacobian of f with respect to x, naturally for certain domains, i.e. the ODE may be stiff in some domains and non-stiff in some other ones.

4.3.6 Adaptive step schemes

In practice adaptive step schemes are used. The idea is very simple if the error is small, i.e. below some tolerance we can take relatively large step h in case of problems, i,e, when the r=error is too large we take much smaller one. The question is how to estimate the error not knowing the real solution? There a few different approaches however the main idea is the same, we get two approximations of the solution at given time t: x_1 and x_2 - the first one much better the other worse, they can be obtained by using schemes of different order or using the same scheme but with different step sizes. Then actually we assume that the better one is as a real solution for the worse one and we can estimate the error for the worse one i.e. we assume that

$$\|x_1(t) - x^*(t)\| \approx \|x_1(t) - x_2(t)\|$$

and we teke then $\|x_1(t) - x_2(t)\|$ as our error estimator.

We will present the idea on two Runge's schems: Heun and explicit Euler.We will estimate the error localy. Let consider IVP:

$$x' = f(t, x) \qquad x(t) = x_n$$

where x_n is a given value then we approximate the solution at the time $t + h$. The step size h is obtained from the previous step (or just given at the first one). We have two tolerances $TOLMIN$ and $TOLMAX$, we want to get the numerical solution such that the error is between those two. In practice we take $TOLMAX = 2 * TOLMIN$ or $3 * TOLMIN$ so the error is on more or less given level.

So we compute two approximations:

$$x1 = x_n + hf(t, x_n) \quad (Euler)$$

and

$$x_2 = x_n + 0.5h(f(t, x_n) + f(t + h, x_n + hf(t, x_n))) \qquad (Heun)$$

Note that actually we should compute first $f_n := f(t, x_n)$ and then

$$\begin{aligned} x_1 &= x_n + hf_n \\ x_2 &= x_n + 0.5h(f_n + f(t + h, x1)) \end{aligned}$$

Thus the main cost is to compute two f evaluation - it is the same cost as using Heun scheme.

Then we compute the error estimator $Est = \|x_1 - x_2\|$ for the Euler scheme If $TOLMIN \leq Est \leq TOLMAX$ then we restart the procedure at $t + h$ with the same

step size h, If $Est \leq TOLMIN$ we decrease the step size to e.g. $2 * h$ and restart the procedure at $t + h$. If $Est \geq TOLMAX$ we decrease h, e.g. to $0.5h$ and recompute the numerical solution starting at the old time t and repeating the estimation.

Naturally in practice we should have some minimal size of the step $hmin$ and a maximal size of the step $hmax < 1$.

That's the simplest adaptivity approach, there also other ones like estimating the size of the right step, or using the same scheme twice with h and e.g. $h0.5$.

Note that we make only assumption that he Heun scheme gives us much better approximation than the Euler scheme. The other observation is less obvious but we have only an estimate of the Euler scheme error, this approach does not give us any information about the Heun scheme error which has to be much less than the Euler one..e big O notation. The last remark is about locality of the estimation, namely we estimate local errors at each step, but we should remember that more or less local error is one order more than the global error in case of the constant step size or order of the scheme, i.e. we can expect that $\|x_1 - x(t)\| = O(h^2)$ for the Euler scheme (naturally with unknown constant in the big O notation...). We should take it into account when we set the tolerances.

Practically we should estimate Euler scheme error but take Heun approximation as a better solution.

5 C language, numerical libraries etc

Using ready systems like matlab or octave is vey convenient, but we have to pay. The codes are slower cannot run on super computers with thousands of processors etc Thus if we want very fast codes we probably have to write codes in lower level languages like C or C++. Naturally we could write everything form scratch but it would take a lot of time, testing also is difficult and our codes still might not be so much faster comparing to e.g. octave which has overhead of the system but use well tested and very efficient numerical packages. Thus it seems the most reasonable to use well tested and optimised libraries whenever it is possible. Usually our numerical code is built from blocks which use well known methods which can be taken form numerical libraries (packages).

5.1 Using numerical fortran libraries in C

Many numerical libraries are pretty old, are being developed for 40 or even more years and were and are written in fortran. Fortunately it is relatively easy to call a fortran function in C (or C++). Only we have to use analogous data types as in fortran.

5.1.1 Real types in fortran and C

we should know hwat real types are in Fortran and their respective counterparts in C.

SO the basic real types in Fortran are:

- REAL this types represents single precision numbers. Its counterpart type in C is float .
- DOUBLE PRECISION this types represents double precision numbers. Its counterpart type in C is double.
- COMPLEX this types represents single precision complex numbers i.e. represented by two single numbers. There is no direct counterpart type in C, but we can represent it as a structure with two float fields.
- COMPLEX$*$16 represents a complex type with two real numbers on 8 bytes - together 16 bytes i.e. two double precision numbers. Again there is no direct counterpart type in C, but again we can use a structure type with two double type fields.

We need also integer and string types:

- INTEGER is a standard integer type, in C can be represented by int type.
- CHARACTER$*$1 represents a character type, in C : char which is one byte integer type.

A real or integer vector is represented as a respective array:

- INTEGER ip $(*)$ declares an integer array, in C we declare: int $*$ip,

- a real vector REAL x (∗) in single precision, in C we have that float ∗x;
- a real double precision vector DOUBLE PRECISION x (∗), in C, we declare double ∗x;

5.1.2 Matrix types in Fortran and C

In C language a matrix is represented as a double arrray: float ∗∗A a pointer to a float pointer. Then an access to an element $A(i, j)$ we get as $A[i][j]$, anyway what is important is that the matrix is stored rowwise in the memory.

But in fortran a matrix is stored column wise as a single array. A standard declaration of a matrix $m \times n$:

```
double precision A (LDA,*)
```

Here LDA is an integer less or equal m (no of rows of A - the length of a column of A). In order to calla fortran function from C we have to represent a matrix a single index array columnwise, i.e. we declare a single real array as

```
float *A;
```

Then we have to allocate memory using e.g. malloc() function and we define elements columnwise e.g. to define the following matrix:

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

we can write in C:

```
for (k=0;k<6;k++){
 A[k]=k+1;
}
```

5.1.3 Prototype

In order to use a fortran function we have to write a proper C function prototype of this fortran function which will be called later by us - as a call to the given fortran function. The general rule is - all parameters of the prototypes must be respective pointers, e.g. if a parameter is an integer type then - a pointer to int etc If a parameter is a vector

of e.g. double precision then a pointer to double type. The naming convention is to use the fortran function name plus an extra underline character i.e. if a fortran function has the name dnrm2() then our prototype name is: dnrm2_(). If this function is real ortran function i.e. it returns a value of a specific type our prototype should be of a respective type, too. If it is a subroutine not returning any value then our prototype should return the void type.

We will present two simple examples: the simplest is the fortran function dnrm2() - which computes the 2nd norm of a double precision vector, its fortran few 1st lines are:

```
      DOUBLE PRECISION FUNCTION DNRM2(N,X,INCX)
*     .. Scalar Arguments ..
      INTEGER INCX,N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION X(*)
*     ..
*
*  Purpose
*  =======
*
*  DNRM2 returns the euclidean norm of a vector via the function
*  name, so that
*
*     DNRM2 := sqrt( x'*x )
*
*
```

The function returns a double precision value and has 3 parameters, two integer type and one double precision vector.

Then our C prototype must return C counterpart of double precision, i.e. double, and has 3 parameters of respective pointer types:

```
double dnrm2_(int *,double *,int *);
```

Then if we want to use this function we have to call it as in prototype, e.g.:

```
nrm=dnrm2_(&N,X,&INCX);
```

naturally $N, INCX$ must be integer variables and X a double array of length equal to the value of N (a pointer to double), $INCX$ may be equal to one to compute the standard

second norm. For other possible values of INCX in a call this function we refer to BLAS documentaion of this function, it can be easily found on-line.

The whole examplary simplest code:

```
  /*let it be a file named blassimplest.c */
#include<stdio.h>
  /*the header file of  input/output functions standard C lib*/

double dnrm2_(int *,double *,int *); /*our prototype*/

void main(void){

  int N=3,INCX=1;
  double X[3],nrm;

  X[0]=X[1]=X[2]=1.0;

  nrm=dnrm2_(&N,X,&INCX);
  /* a call to the fortran function as in prototype*/


  printf("2nd norm of [%g;%g;%g]=%g\n",X[0],X[1],X[2],nrm);
  return;
}/*end of main()*/
```

Naturally, we have to compile it - in linux gcc is GNU compiler collection and contains both C and fortran ocompilers so it is enough to write gcc blassimplest.c −lblas , here −lblas means that the compiler will use the library blas, our fortran function is a part of this library.

If we have an older linux system we may need some extra libraries as −lgfortran or −lg2c, or even −lm - the last command says that our program uses math library when one can find a square root function necessary for computing the 2nd norm. Anyway we also assume that blas lib is installed, which should be easy to do in linux, it is usually a part of standard distributions. If not, it should be in any distribution as a package and can be easily installed. We will need it later anyway.

If all fails one can download dnrm2.f fortran source file and just compile it using e.g. the following command: gcc −c dnrm2.f to the object file: dnrm2.o which can be later linked with our C file:

```
gcc name.c dnrm2.o
```

where name.c is our source file with the function main().

The next example is the following fortran function SGEMV() which is again a BLAS level 2 function. BLAS level 2 functions contains functions computing matrix-vector basic operations of linear algebra. The naming conventions of BLASfunction we will explain later in one of following sections. Here it is enough to know that the first letter means precision, i.e. *s* single (REAL in fortran, float type in C) and *d* double one (DOUBLE PRECISION type in fortran and double type in C).

The first few lines of the source of this function are the following:

```
      SUBROUTINE SGEMV(TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
*     .. Scalar Arguments ..
      REAL ALPHA,BETA
      INTEGER INCX,INCY,LDA,M,N
      CHARACTER TRANS
*     ..
*     .. Array Arguments ..
      REAL A(LDA,*),X(*),Y(*)
*     ..
*
*  Purpose
*  =======
*
*  SGEMV  performs one of the matrix-vector operations
*
*     y := alpha*A*x + beta*y,   or   y := alpha*A'*x + beta*y,
*
*  where alpha and beta are scalars, x and y are vectors and A is an
*  m by n matrix.
*
*  Arguments
*  ==========
*
*  TRANS  - CHARACTER*1.
*           On entry, TRANS specifies the operation to be performed as
*           follows:
*
*              TRANS = 'N' or 'n'   y := alpha*A*x + beta*y.
*
*              TRANS = 'T' or 't'   y := alpha*A'*x + beta*y.
*
```

```
*              TRANS = 'C' or 'c'   y := alpha*A'*x + beta*y.
*
*           Unchanged on exit.
*
*  M      - INTEGER.
*           On entry, M specifies the number of rows of the matrix A.
*           M must be at least zero.
*           Unchanged on exit.
*
*  N      - INTEGER.
*           On entry, N specifies the number of columns of the matrix A.
*           N must be at least zero.
*           Unchanged on exit.
*
*  ALPHA  - REAL             .
*           On entry, ALPHA specifies the scalar alpha.
*           Unchanged on exit.
*
*  A      - REAL             array of DIMENSION ( LDA, n ).
*           Before entry, the leading m by n part of the array A must
*           contain the matrix of coefficients.
*           Unchanged on exit.
*
*  LDA    - INTEGER.
*           On entry, LDA specifies the first dimension of A as declared
*           in the calling (sub) program. LDA must be at least
*           max( 1, m ).
*           Unchanged on exit.
*
*  X      - REAL             array of DIMENSION at least
*           ( 1 + ( n - 1 )*abs( INCX ) ) when TRANS = 'N' or 'n'
*           and at least
*           ( 1 + ( m - 1 )*abs( INCX ) ) otherwise.
*           Before entry, the incremented array X must contain the
*           vector x.
*           Unchanged on exit.
*
*  INCX   - INTEGER.
*           On entry, INCX specifies the increment for the elements of
*           X. INCX must not be zero.
*           Unchanged on exit.
*
*  BETA   - REAL             .
```

```
*           On entry, BETA specifies the scalar beta. When BETA is
*           supplied as zero then Y need not be set on input.
*           Unchanged on exit.
*
*  Y      - REAL             array of DIMENSION at least
*           ( 1 + ( m - 1 )*abs( INCY ) ) when TRANS = 'N' or 'n'
*           and at least
*           ( 1 + ( n - 1 )*abs( INCY ) ) otherwise.
*           Before entry with BETA non-zero, the incremented array Y
*           must contain the vector y. On exit, Y is overwritten by the
*           updated vector y.
*
*  INCY   - INTEGER.
*           On entry, INCY specifies the increment for the elements of
*           Y. INCY must not be zero.
*           Unchanged on exit.
*
*
*  Level 2 Blas routine.
```

The function computes $y = \alpha * Ax + \beta * y$ or $y = \alpha * A^T x + \beta * y$, where α, β are scalars, x, y a vector, A matrix, the result is given in the vector y (so its value is overwritten!). Thus we the C prototype might look as follows:

```
void rgemv_(int *);
```

Naturally if called the respective variables must be properly declared, memory (in case of dynamical allocation) allocated etc The returned type void means that the function does nor return any value. Below we present a very simple C code in which we multiply a vector by a matrix.

```
/*file name : exAx.c */
#include<stdio.h>
void sgemv_(char*,int*,int*,float*,float*,int*,
              float*,int*,float*,float*,int*);
void main(void){
 float alpha=1.0,beta=0.0; /* single prec scalars */

 char trans='N';

 int M=3,N=2,INCX=1,INCY=1,LDA=M;

 float x[2],y[3]; /*vectors*/
```

```
  float A[6]; /*our matrix 3x2 kept as a float vector */

   int k;

/*we have to define the values of entries of  A and vectors x ,y*/
   for(k=0;k<6;k++)
     A[k]=k+1;

   x[0]=x[1]=1.0;
   y[0]=y[1]=y[2]=0.0;

   /*we compute y=A*x */
   sgemv_(&trans,&M,&N,&alpha,A,&LDA,x,&INCX,&beta,y,&INCY);

   printf("y=A[%g;%g]=[%g;%g;%g]\n",x[0],x[1],y[0],y[1],y[2]);

 return;
}/* main()*/
```

Then if we execute gcc −c exAx.c then the precompiler will find syntax errors or create an object file exAx.o which can further linked with BLAS library. ebra libraries: BLAS and LAPACK.

5.2 Numerical libraries

We will discuss in details two linear algebra libraries: BLAS and LAPACK.

Both comprise fortran functions: BLAS solving simple linear algebra operations which are blocks of any more serious linear algebra solvers which contains LAPACK. More details below.

5.2.1 Basic Linear Algebra Subprograms (BLAS)

This fortran library contains fortran functions with absolutely basic operations of linear algebra like adding scaled vectors, standard scalar product, second norm of a vector, multiplication of vector by a matrix, or multiplication of two matrices - that's the most complicated. Anyway all operations are very simple but they are crucial blocks in any

more complicated linear algebra solver. If they are executed fast i.e. BLAS functions are executed in optimal way - we say that we have optimized BLAS, then the more complicated solvers work also much faster.

The function of BLAS lib are divided into 3 levels:

1. Level 1: basic vector operations like a 2nd norm of a vector, a scaling of a vector etc, e.g.
   - xNRM2 - computes the 2nd norm of a vector: $nrm2 \leftarrow \|x\|_2$.
   - xSCAL - scales a vector $\vec{x} \leftarrow \alpha * \vec{x}$.
   - xAXPY - computes generalized vector addition
   $$\vec{y} \leftarrow \alpha * \vec{x} + \vec{y}$$
2. Level 2: matrix vector operations e.g. multiplication of a vector by a matrix
   - xGEMV - computes generalized matrix vector multiplication
   $$\vec{y} \leftarrow \alpha * A * \vec{x} + \beta * \vec{y}$$
   or
   $$\vec{y} \leftarrow \alpha * A^T * \vec{x} + \beta * \vec{y}$$
   or
   $$\vec{y} \leftarrow \alpha * A^H * \vec{x} + \beta * \vec{y}$$
   The last version works for complex precisions.
   - xGER - creating a rank one modified matrix:
   $$A \leftarrow \alpha * \vec{x}\vec{y}^T + A$$
3. Level 3: matrix matrix operations like multiplication of two matrices, e.g.
   - xGEMM - computes generalized matrix matrix multiplication
   $$C \leftarrow \alpha * op(A) * op(B) + \beta * C$$
   where $op(X) = X, X^T, X^H$. The last version works for complex precisions.
   - xSYRK - computes generalized matrix matrix multiplication
   $$C \leftarrow \alpha * A * A^T + \beta * C$$
   or
   $$C \leftarrow \alpha * A^T * A + \beta * C$$
   where C is $n \times n$ matrix.

The prefix x in the name means precision for which the function works.

The naming scheme of BLAS library is the following: the name of a function is either xZZZ in case of vector operations or xYYZ(ZZ) where x is the precision of the function (type of floting point variables), Z(ZZ) is the name of the operation, e.g. NRM2 in the function DNRM2 means the second norm, and YY is the type of a matrix, e.g. GE in DGEMV is GEneral type - any full matrix, the prefix D means double precision, and MV is matrix times vector operation. There are four precisions prefixes in BLAS (and LAPACK as well):

1. S - single precision, i.e. real variables
2. D - double presicion
3. C - complex single precision
4. Z - complex double presicion

Not all functions work for all precisions. Thus DNRM2 computes 2nd norm of a real double precision vector, while CNRM2 computes 2nd norm of a complex single precision vector (each entry is single precision complex number).

5.2.2 Linear Algebra PACKage (LAPACK)

The LAPACK fortran library comprise a selection of linear algebra solvers.

The naming scheme is similar to BLAS namely: the name of a function is xYYZZ(Z) where x is the precision of the function (type of floating point variables), YY is the type of the matrix, ZZ(Z) is the name of the problem or method, and YY is the type of a matrix, e.g. in DGESVX - D is a prefix menaing double precision, GE is GEneral type - any full matrix, SVX means solving linear system - a 'for expert' routine.

There are the same four precisions prefixes in LAPACK, same as in BLAS, see above.

The function are divided into 3 classes:

- **driver routines**: solving major computational problems of linear algebra, e.g computing eigenvalue, solving a linear system etc

- **computational routines**: computing the main computational tasks, e.g. LU decomposition of a matrix, which can be calld by a driver routine solving a linear system of equations. In general an user or specially a developer may need them to wite a driver type routine. A driver routine call usually a sequence of computational routines.
- **auxiliary routines**: BLAS extensions, routines performing block operations, some low level computations like e.g. scaling of a matrix etc

We have the following types of matrices (YY in the name of a function)

- BD bidiagonal
- DI diagonal
- GB general band
- GE general (i.e., unsymmetric, in some cases rectangular)
- GG general matrices, generalized problem (i.e., a pair of general matrices)
- GT general tridiagonal
- HB (complex) Hermitian band
- HE (complex) Hermitian
- HG upper Hessenberg matrix, generalized problem (i.e a Hessenberg and a triangular matrix)
- HP (complex) Hermitian, packed storage
- HS upper Hessenberg
- OP (real) orthogonal, packed storage
- OR (real) orthogonal
- PB symmetric or Hermitian positive definite band
- PO symmetric or Hermitian positive definite
- PP symmetric or Hermitian positive definite, packed storage
- PT symmetric or Hermitian positive definite tridiagonal
- SB (real) symmetric band

- SP symmetric, packed storage
- ST (real) symmetric tridiagonal
- SY symmetric
- TB triangular band
- TG triangular matrices, generalized problem (i.e., a pair of triangular matrices)
- TP triangular, packed storage
- TR triangular (or in some cases quasi-triangular)
- TZ trapezoidal
- UN (complex) unitary
- UP (complex) unitary, packed storage

Next we present main computational problems which are solved by LAPACK routines:

- solving linear equations: a simple driver i.e. ending with SV and expert drivers ending with SVX, e.g.:
  - : xGESV -solving general matrices linear system
  - : xGESVX which can also solve $A^T X = B$ or $A^H X = B$, check for near singularity, estimate of the pivot growth, estimate for the condition number of the matrix, refine the solution, estimate forward or backward errors etc
- Linear Least Square (LLS) Problem - xGELSX, xGELSY, xGELSS, and xGELSD, solve LLSP, even if it happens that A is rank-deficient; xGELSX and xGELSY are based on a complete orthogonal factorization of A, xGELSS and xGELSD utilize the singular value decomposition (SVD) of A, but xGELSD uses SVD computed by an algorithm which is divide and conquer.
  - xGELSX -retained for compability with older LAPACK versions
  - xGELSY faster than xGELSX but requires more workspace - it calls a block algorithm to perform the orthogonal factorization.
  - xGELSD is faster than xGELSS but may require a tiny bit larger workspace
- Generalized Linear Least Squares (LSE and GLM) Problems

– linear equality-constrained least squares problem (LSE) is a problem to find:

$$\arg\min_x \|Ax - f\|_2 \quad \text{subject} \quad \text{to} \quad Bx = d$$

If A is $m \times n$ and B is $p \times n$ matrix, with $p \leq n \leq m + p$ then the problem has a unique solution under the assumption that B has a full row rank, and

$$\begin{pmatrix} A \\ B \end{pmatrix}$$

has full column rank n. Then, the routine xGGLSE finds the solution using the generalized RG (GRQ) factorization.

– A general (Gauss-Markov) liner model problem (GLM) is the second type of generalized linear lest square problem. We want to find y and x

$$\arg\min_x \|y\|_2 \quad \text{subject} \quad \text{to} \quad d = Ax + By$$

where A is an $n \times m$ matrix, B is an $n \times p$ matrix, with $m \leq n \leq m + p$. The problem has unique solutions x and y if A has full column rank m, and the matrix

$$\begin{pmatrix} A & B \end{pmatrix}$$

has full row rank n. The routine xGGGLM finds the solutions using the generalized QR (GQR) factorization.

- Symmetric eigenvalue problem (SEP) -matrix is symmetric (if real) or hermitian (if complex)

  – A simple driver (name ending -EV) computes all eigenvalues and as an option eigenvectors, e.g. SSYEV, CHEEV, DSYEV, ZHEEV.

  – AN expert driver (nam ending -EVX) solves the same problem, but can also compute a selected cluster of the eigenvalues (and optionally eigenvectors) and is much faster then, e.g. DSYEVX. -A divide-and-conquer driver (name ending -EVD) works as a simple driver but is much faster for larger matrices than the simple one but requires larger workspace. The name comes from the algorithm, e.g. DSYEVD.

  -A relatively robust representation (RRR) driver (name ends as -EVR) computes all or a subset of eigenvalues (optionally eigenvectors) and it is usually the fastes algorithm of all and uses the smallest workspace, e.g. DSYEVR.

- Nonsymmetric Eigenproblems (NEP) - the matrix can be nonsymmetric or non-hermitian (if complex). In this case if A the matrix is real then it may have a conjugate complex pair of eigenvalues. The problem may be solved by the Schur factorization:

$$A = QUQ^T$$

where Q is an orthogonal matrix, U is an upper quasi-triangular matrix with 1×1 and 2×2 diagonal blocks (in case of all real eigenvalues the matrix is real upper triangular). If A complex

$$A = QUQ^H$$

where Q is a unitary matrix $Q^HQ = I$, and U is a complex upper triangular. The columns of Q are the Schur vectors of A.

- A simple driver xGEES computes all or part of the Schur factorization of A, optionally the eigenvalues are ordered.
- An expert driver xGEESX which can also additionally compute condition numbers for the average of some selected eigenvalues and the corresponding invariant subspace.
- A simple driver xGEEV computes all eigenvalues and optionally right or left eigenvectors.
- An expert driver xGEEVX can additionally balance the matrix in order to improve the conditioning of the eigenapairs, it also computes the condition numbers of the eigenvalues and left or right eigenvectors.

- Singular Value Decomposition (SVD)

  SVD of any matrix A is given as

  $$A = Q\Sigma V^T$$

  where Q, V orthogonal matrices, Σ non-negative diagonal one, or

  $$A = Q\Sigma V^H$$

  if A complex, then Q, V unitary.

  - xGESVD a simple driver computes all singular values -diagonal of the diagonal matrix $\Sigma$and optionally left (columns of Q) or right (columns of V) singular vectors (SVD: $A = Q\Sigma V^T$ or $AV = Q\Sigma$ or $A^TQ = V\Sigma$)
  - xGESDD a divide-and-conquer driver computes the same result as the simple driver but is much faster for large matrices but utilizes larger workspace. Underlying algorithm is a divide-and-conquer one.

- Generalized Eigenvalue and Singular Value Problems

  - Generalized Symmetric Definite Eigenproblems (GSEP), we want to compute:
    1. $Ax = \lambda Bx$
    2. $ABz = \lambda z$
    3. $BAz = \lambda z$,

A, B are symmetric or Hermitian and B is positive definite. Simple Drivers e.g. xSYGV (real), xHEGV (complex).

- Generalized Nonsymmetric Eigenproblems (GNEP), e.g. simple drivers xGGES, (Schur factorization) or xGGEV for eigenvalues/eigenvectors.
- Generalized Singular Value Decomposition (GSVD), e.g. a driver xGGSVD.

As a data we have here two matrices A and B and we want to find generalized eigenpairs: $A\vec{v} = \lambda B\vec{v}$ in GSEP or GNEP or a pair of factorizations in case of GSVD.

5.2.3 Other numerical libraries

We just list the names of some libraries:

- eigenvalue problem for sparse matrices: ARPACK
- ode solving libraries: e.g. lsode lib, cvode, odeint etc
- nonlinear solvers e.g. kinsol, minpack, petsc etc
- sparse numerical linear algebra packages; sparse direct solver e.g. umfpack, sparse,
- iterative linear solvers e.g. Petsc (actually just a part of PETSC contains linear iterative solvers and preconditioners), hypre etc
- PDEs - PETSC - Portable, Extensible Toolkit for Scientific Computation (PETSc), is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations.
- general numerical libs NAG (many languages in particular C,Fortran,C++) , GNU scientific library, Harwell Subroutine Library (fortran) etc
- FFTW- (FFT in the West) fast fourier and related transforms ©
- QUADPACK- numerical integration in one dimension (octave use it in the function quad())
- Intel MKL, Intel Math Kernel Library (in C), a library of optimized math routines for science, engineering, and financial applications, written in C/C++ and Fortran. Core math functions include BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math.

6 Solving boundary value problems in 1D

We will consider a model linear boundary value problem:

$$\begin{aligned} -u'' + d(t)u' + c(t)u &= f(t) \qquad t \in (a,b) && (1)\\ u(a) &= \alpha, && (2)\\ u(b) &= \beta. && (3) \end{aligned}$$

The first equation is a linear 2nd order scalar equation, and two next two lines are Dirichlet boundary conditions.

This problem may have no solutions, i.e. the ODE always have a so called general solution, the Initial Value Problem (IVP) :

$$\begin{aligned} -u'' + d(t)u' + c(t)u &= f(t) \qquad t \in (a,b) && (4)\\ u(a) &= \alpha, && (5)\\ u'(a) &= \nu. && (6) \end{aligned}$$

also has a unique solution.

THe simples example is:

$$\begin{aligned} -u'' - u &= 0 \qquad t \in (a,b) && (7)\\ u(0) &= 1, && (8)\\ u(\pi) &= 0. && (9) \end{aligned}$$

The general solution of this ODE:

$$u_g(t) = \gamma_1 \sin(x) + \gamma_2 \cos(x)$$

thus we see that the boundary values give us two equations $u_g(0) = \gamma_1 = 1$ and $u_g(\pi) = -\gamma_1 = 0$ which cannot be satisfied. Even if there is a solution it canno be unique e.g. if $\alpha = -\beta = 1$ since $u(t) = \cos(t) + \gamma_2 \sin(t)$ for any γ_2.

Try to find values of d, c, a, b (Assuming that $d(t), c(t)$ are constant) such that

$$\begin{aligned} -u'' + du' + cu &= 0 \qquad t \in (a,b) && (10)\\ u(a) &= \alpha, && (11)\\ u(b) &= \beta. && (12) \end{aligned}$$

has a unique solution for any α, β.

6.1 Shooting method

If there is a unique solution od BVP, then this solution solves Initial Value Problem (IVP) with the same ODE but with the following initial values:

$$\begin{aligned} -u'' + d(t)u' + c(t)u &= f(t) \qquad t \in (a,b) && (13)\\ u(a) &= \alpha, && (14)\\ u'(a) &= s_0 && (15) \end{aligned}$$

The shooting method more or less is based on finding such s_0 that the solution of IVP with s_0: $u(t, s_0)$ solves the BVP.

How to do it...

In linear case it is simple just solve IVP for two different s, e.g. $s_1 = 0$ and $s_2 = 1$ getting $u(t, s_1)$ and $u(t, s_2)$, in particular we get $u(b, 0)$ and $u(b, 1)$ then the solution is of the form: (all of such IVP - theory of ODE)

$$u(t, s) = u_0(t) + s * u_1(s) \tag{16}$$

Then substituting at $t = b$ we get

$$u(b, 0) = u_0(b) \quad u(b, 1) = u_0(b) + u_1(b)$$

i.e.

$$u_1(b) = u(b, 1) - u(b, 0).$$

Now we get s_0 solving a liner scalar equation:

$$\beta = u(b, s_0) = u_0(b) + s_0 * u_1(b)$$

getting

$$s_0 = (be - u_1(b))/u_1(b) = (\beta - u(b, 0))/(u(b, 1) - u(b, 0)).$$

Naturally, $u(b, 0)$ and $u(b, 1)$ are computed by some numerical approximated scheme. In case of the the coefficients $d(t), c(t)$ which are not constant we may not know if there is a unique solution, but (16) is true. Thus if computed $u_1(b) = u(b, 1) - u(b, 0)$ is non-zero we know that there is a unique s_0 such that IVP with that s_0 solves our BVP, thus we can practically check if we have a unique solution. Since in practise we cannot usually get exact values but approximations. Thus in practise we compute an approximation of $u_1 b(b)$ as above, and the if this value is above some level (depending on the fl accuracy and our ODE solver accuracy) we may assume that there is an unique solution, otherwise even if our approximation is nonzero and formally we can compute s_0 we cannot be sure

if the real solution exits... If we write code then we may issue a worning to the output e.g. show a warning on a display.

In a more complicated nonlinear case e.g.

$$\begin{aligned} u'' &= F(t,u,u') \qquad t \in (a,b) && (17)\\ u(a) &= \alpha, && (18)\\ u(b) &= \beta. && (19) \end{aligned}$$

we may do the same, namely, pose an IVP for the same ODE, same left boundary value (we could also use the right one and integrate the ODE backward...) and try to find the right initial value of the derivative:

$$\begin{aligned} u'' &= F(t,u,u') \qquad t \in (a,b) && (20)\\ u(a) &= \alpha, && (21)\\ u'(a) &= s. && (22) \end{aligned}$$

That is find s_0 such that the solution of this IVP with $u'(a) = s_0$ solves our BVP.

How to find numerically this s_0?

Again refering to general theory of ODEs we know that the solution of IVP: $u(t;b)$ is as smooth as the function F. So really we want to solve a nonlinear univariate equation:

$$g(s) := u(b;s) = \beta.$$

If F is C^1 smooth then g is also C^1 function and we can apply any nonlinear solver e.g. the Newton method, or a bisection method if we find two s_1, s_2 such that $g(s_1) * g(s_2) < 0$.

Note that (20) may be multidimensional, i.e. $u(t) \in R^m$ and then we get a system of nonlinear equations $g(s) = \beta \in R^m$, and we have apply a nonlinear solver for multi-dimensional nonlinear system of equations, i.e .in practise we apply some ODE solver in order to compute $g(s)$ for any s. If we want to use the Newton or similar method we have to know how to compute the value (or its aproximation) of $\frac{\partial g}{\partial s}$. Again to do it we

may apply an ODE solver (or ratehr IVP solver) to

$$
\begin{aligned}
u'' &= F(t,u,u') \qquad t \in (a,b) && (23)\\
\frac{\partial u''}{\partial s} &= \frac{\partial F}{\partial u}(t,u,u')\frac{\partial u}{\partial s} + \frac{\partial F}{\partial u'}(t,u,u')\frac{\partial u'}{\partial s} \quad t \in (a,b) && (24)\\
u(a) &= \alpha, && (25)\\
u'(a) &= s && (26)\\
\frac{\partial u}{\partial s} &= 0 && (27)\\
\frac{\partial u'}{\partial s} &= Id && (28)\\
& && (29)
\end{aligned}
$$

Note, that this ODE system is $m + m * m$ dimensional... however is linear with respect to $\frac{\partial u}{\partial s}$. Instead of the standard Newton method we may use its version wit hthe Jacobian approximated by finite differences or some other nonlinear solver. Whatever method we use we always have to compute the solutions of IVP for iterations in s we are sometimes called "shoots", the name of the method comes form this.

It may also happen that the boundary values conditions are nonlinear e.g.:

$$
\begin{aligned}
u'' &= F(t,u,u') \qquad t \in (a,b) && (30)\\
u(a) &= \alpha, && (31)\\
f(u(b)) &= 0. && (32)
\end{aligned}
$$

for some nonlinear function $f : D \subset R^m \to R^m$. Then our nonlinear equation or system of equations is just:

$$g(s) := f(u(b;s)) = 0,$$

it is nonlinear even if the ODE is linear.

6.2 Finite Difference Method (FDM) in 1D

We want again to solve a model linear problem:

$$
\begin{aligned}
-u'' + cu &= f \qquad t \in (a,b), && (33)\\
u(a) &= \alpha, && (34)\\
u(b) &= \beta. && (35)
\end{aligned}
$$

where $c(x)$ is nonengaive continuous function.

The second derivative may be approximated by three point finite difference:

$$\partial\overline{\partial}u(x) = \overline{\partial}\partial u(x) = \frac{u(x-h) - 2u(x) + u(x+h)}{h^2} \approx u''(x)$$

which is the composition of the forward difference

$$\partial_h u(x) = \frac{u(x+h) - u(x)}{h} \approx u'(x),$$

and the backward difference:

$$\overline{\partial}_h u(x) = \frac{u(x) - u(x-h)}{h} \approx u'(x).$$

If u sufficiently smooth we have that:

$$\partial\overline{\partial}u(x) = \overline{\partial}\partial u(x) = \frac{u(x-h) - 2u(x) + u(x+h)}{h^2} = u''(x) + O(h^2).$$

In finite difference method we have to introduce a mesh in $[a, b]$, the simplest here is the equidistantly spaced one:

$$x_k^h = a + b * h \quad k = 0, \ldots, N \quad h = \frac{b-a}{N}.$$

Then we pose our discrete problem on the interior mesh points replacing the derivative by its finite difference approximation and getting the following discrete problem: find $u_h : \{x_k^h\}_{k=0}^N \to R$

$$\begin{aligned} u_h(x_0) &= \alpha \\ \frac{-u_h(x_k - h) + 2u_h(x_k) - u_h(x_k + h)}{h^2} + c(x_k)u_h(x_k) &= f(x_k) \quad k = 1, \ldots, N-1 \\ u_h(x_N) &= \beta \end{aligned}$$

If we introduce a simpler notations:

$$u_k := u_k^h := u_h(x_k) \quad k = 0, \ldots, N, \qquad F_0 = \alpha,\ F_k := f(x_k),\ k = 1, \ldots, N-1,\ F_N = \beta,$$

we get the following system:

$$\begin{aligned} u_0 = F_0 &= \alpha, \\ \frac{-u_{k-1} + 2u_k - u_{k+1}}{h^2} + c(x_k)u_k &= F_k \quad k = 1, \ldots, N-1, \\ u_N = F_N &= \beta. \end{aligned} \tag{36}$$

The 1st and the last equations define $u_0 = \alpha, u_N = \beta$ so we can reduce the system into:

$$\begin{aligned}
\frac{2u_1 - u_2}{h^2} + c(x_1)u_1 &= F_1 + \frac{\alpha}{h^2}, \\
\frac{-u_{k-1} + 2u_k - u_{k+1}}{h^2} + c(x_k)u_k &= F_k \quad k = 2, \ldots, N-2, \\
\frac{-u_{N-2} + 2u_{N-1}}{h^2} + c(u_{N-1})u_{N-1} &= F_{N-1} + \frac{\beta}{h^2}.
\end{aligned}$$

This is a system of the form:

$$\left(\frac{1}{h^2}A + C\right)\hat{u} = \hat{F}$$

where

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}, \qquad C = \begin{pmatrix} c(x_1) & & & & \\ & c(x_2) & & & \\ & & \ddots & & \\ & & & c(x_{N-2}) & \\ & & & & c(x_{N-1}) \end{pmatrix},$$

i.e. A is tridiagonal symmetric matrix of constant diagonals (having two on the main one, and minus one on the super-diagonal and sub-diagonal), C is a diagonal matrix, which has the main diagonal vector equal to $(c(x_1), c(x_2), \ldots, c(x_{N-1}))^T$, and

$$\hat{F} = \begin{pmatrix} F_1 + \frac{\alpha}{h^2} \\ F_2 \\ \ldots \\ F_{N-2} \\ F_{N-1} + \frac{\beta}{h^2} \end{pmatrix}, \qquad \hat{u} = \begin{pmatrix} u_1 \\ u_2 \\ \ldots \\ u_{N-2} \\ u_{N-1} \end{pmatrix}$$

i.e the vector $\hat{u}$ is our discrete solution restricted to interior mesh points.

We can also solve the original whole system (36) which is also a tridiagonal linear system.

Both systems can solved by a direct solver for banded systems e.g. the version of LU factorization (Gauss elimination) in O(N) operations. In octave we should define the matrix of the system in sparse format and just apply backslash u=B\F, where B is the matrix of the system and F the rhs, then octave will use its sparse direct solver.

If we plug in the solutions into the scheme we get (please, recall $-u''(x_k)+c(x_k)u(x_k) = f(x_k)$ for $x_k \in (a, b)$)

$$
\begin{aligned}
|u(x_0) - \alpha| &= 0, \\
\left| \frac{-u(x_k - h) + 2u_h(x_k) - u(x_k + h)}{h^2} + cu(x_k) - f(x_k) \right| &= O(h^2) \quad k = 1, \ldots, N-1, \\
|u(x_N) - \beta| &= 0.
\end{aligned}
$$

Taking maximum over all we get the so called local truncation error which is like $O(h^2)$, and we can say (not defining it precisely) that the scheme is of the second order.

We will not discuss the convergence theory of the FDM method, it is enough if you know that

$$\max_k |u(x_k^h) - u_k^h| = O(h^2)$$

as long as the u the solution of the original BVP is smooth enough. We say that the convergence in the discrete maximum norm is of the second order.

6.2.1 Neumann and Robin boundary condition in 1D

. We want again to solve a model linear problem but with different boundary condition:

$$
\begin{aligned}
-u'' + cu &= f \qquad t \in (a, b), && (37)\\
u(a) &= \alpha, && (38)\\
u'(b) &= \beta \qquad (Neumann). && (39)
\end{aligned}
$$

where $c(x)$ is nonengaive continuous function or

$$
\begin{aligned}
-u'' + cu &= f \qquad t \in (a, b), && (40)\\
u(a) &= \alpha, && (41)\\
u'(b) + d * u(b) &= \beta \qquad (Robin). && (42)
\end{aligned}
$$

We will focus on the Neumann condition, then in any FDM scheme we have to approximate somehow the derivative at b by finite differences defined by the values of the FDM discrete function at the mesh point. In our example it is natural to use the backward difference on two last mesh points:

$$\overline{\partial} u_h(x_N) = \beta.$$

so we get the following FDM scheme: find $u_h : \{x_k^h\}_{k=0}^N \to R$

$$\begin{aligned} u_h(x_0) &= \alpha \\ \frac{-u_h(x_k-h)+2u_h(x_k)-u_h(x_k+h)}{h^2} + c(x_k)u_h(x_k) &= f(x_k) \quad k=1,\ldots,N-1 \\ \overline{\partial} u_h(x_N) &= \beta \end{aligned}$$

Everything looks fine, but it is only first order scheme, if we substitute the solution to the scheme and shift everything to the left side we get $O(h)$ in the last equation since:

$$\overline{\partial} u(x) - u'(x) = O(h).$$

The queston arises if we can change the FDM scheme in order t o preserve the 2nd order of it?

The answer is YES and there are different methods, namely:

1. Adding a ghostpoint at the right end. If we assume that our equation can or precisely the coefficients function are continuous up to the right end b and u can be extended smoothly outside the interval we can add a ghostpoint: $x_{N+1} = b+h$ and replacing the backward diffirence at b by the central one what gives us a different FDM scheme last equation:

   $$\frac{u_{N+1}-u_{N-1}}{2h} = \beta.$$

   Naturally, then we get an extra unknown - the value of u_h at the ghostpoint u_{N+1} so we need an extra FDM scheme equation: we approximate

   $$-u'' + c(b)u(b) = f(b)$$

   by the 3 point stencil getting:

   $$\frac{-u_h(x_N-h)+2u_h(x_N)-u_h(x_N+h)}{h^2} + c(x_N)u_h(x_N) = f(x_N)$$

   Finally givig us the following FDM scheme:

   $$\begin{aligned} u_h(x_0) &= \alpha \\ \frac{-u_h(x_k-h)+2u_h(x_k)-u_h(x_k+h)}{h^2} + c(x_k)u_h(x_k) &= f(x_k) \quad k=1,\ldots,N \\ \frac{u_{N+1}-u_{N-1}}{2h} &= \beta. \end{aligned}$$

   This a 2nd order scheme.

2. Using the differential equation at b UNder same assumption as in the previous item, i.e .that u, f, c can be exended right we note that if we substitute the solution to the backward difference (the lst FDM equaion) we get that

$$\overline{\partial}u(b) - u'(b) = -0.5u''(b)h + O(h^2).$$

But using the differential equation wehave that:

$$-u''(b) = f(b) - c(b)u(b)$$

Substituting this to the previous eqution we get:

$$\overline{\partial}u(b) - u'(b) = 0.5h(f(b) - c(b)u(b)) + O(h^2).$$

so

$$\overline{\partial}u(b) - \beta - 0.5h(f(b) - c(b)u(b)) = O(h^2).$$

Using this we note that if we replace the backward difference from ourt scheme by the following FDM equation:

$$\overline{\partial}u_h(b) - \beta - 0.5h(f(b) - c(b)u_h(b)) = 0$$

or

$$\overline{\partial}u_h(b) - 0.5hc(b)u_h(b) = beta + 0.5hf(b)$$

we get a second order scheme:

$$\begin{aligned} u_h(x_0) &= \alpha \\ \frac{-u_h(x_k - h) + 2u_h(x_k) - u_h(x_k + h)}{h^2} + c(x_k)u_h(x_k) &= f(x_k) \quad k = 1, \ldots, N-1 \\ \overline{\partial}u_h(b) + 0.5hc(b)u_h(b) &= \beta + 0.5hf(b). \end{aligned}$$

3. The last the simplest approach is to approximate the derivative at b on the 3 point stencil $b-2h, b-h, b$ by a 2nd order finite difference getting a FDM scheme. Thus only one the last equation in the FDM scheme is changed without any extra assumptions. The details are left for a reader.

6.2.2 The pure Neumann condition

In case we have the following BVP:

$$\begin{aligned} -u'' &= f \qquad t \in (a, b), && (43) \\ -u'(a) &= \alpha, \quad (Neumann) && (44) \\ u'(b) &= \beta \qquad (Neumann). && (45) \end{aligned}$$

The solution exists if the following compability condition $\alpha + \int_a^b f dx + \beta = 0$ is satisfied, but we have no uniqueness - it is obvious for any solution u we see that $u + c$ is also a solution with c any constant. (f, α and β also must satisfy special conditions to get a solution at all).

Consider then FDM scheme (for simplicity with low order approximation)

$$\begin{aligned} \partial_h u_h(x_0) &= \alpha \\ \frac{-u_h(x_k - h) + 2u_h(x_k) - u_h(x_k + h)}{h^2} &= f(x_k) \quad k = 1, \ldots, N-1 \\ \overline{\partial}_h u_h(b) &= \beta. \end{aligned}$$

which can be rescaled into:

$$\begin{aligned} \partial_h u_h(x_0) &= \alpha \\ \frac{-u_h(x_k - h) + 2u_h(x_k) - u_h(x_k + h)}{h} &= h * f(x_k) \quad k = 1, \ldots, N-1 \\ \overline{\partial}_h u_h(b) &= \beta. \end{aligned}$$

and then rewritten into the linear system:

$$A\vec{u} = \vec{F}$$

with $A = A^T$ - one can also show (not straightforward - a problem) that $A \geq 0$. We know the kernel of A, namely, $Ker(A) = Span(\vec{1})$ with $\vec{1}$ the constant vector with value one. (Problem: show that kernel has indeed the dimension equal to one). Thus we represent each vector as: $\vec{u} = u_0\vec{1} + (0, u_1 - u_0, \ldots, u_N - u_0)$. Considering our system in a new basis: $(\vec{1}, \vec{e}_1, \ldots, \vec{e}_N)$ i.e. representing $\vec{u} = C\vec{v}$ and multiplying the system by C^T we get that

$$C^T ACv = C^T \vec{F} =: \vec{G}$$

and the matrix in a block form:

$$C^T AC = \begin{pmatrix} 0 & 0 \\ 0 & A(1:N, 1:N) \end{pmatrix}$$

If $G_0 = 0$ i.e. $\vec{1}^T F = \alpha + h\sum_{k=1}^{N-1} f(x_k) + \beta = 0$, the system has a solution unique up to the value v_0 whichcan be defined arbitrarily. Note that the term $h\sum_{k=1}^{N-1} f(x_k)$ approximates (not too well) $\int_a^b f(x)dx$ so our discrete compability condition is a kind of approximation of the original one for the BVP. Please, recall that this FDM scheme of the very low order, so this approximation is very weak.

Then the new problem has a unique solution for its unknown no: $1, \ldots, N$ - the solution of the following linear system:

$$A(1:N, 1:N) \begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} = \begin{pmatrix} G_1 \\ \vdots \\ G_N \end{pmatrix}. \tag{46}$$

The matrix of the system $A(1:N, 1:N)$ as a minor - a submatrix of A is symmetric, tridiagonal and non-negative definite, and it can be shown that also positive definite (a problem) so ti can be solved by a proper version of Cholesky decomposition method with the cost $O(N)$.

- We check if $G_0 = \vec{1}^T F = \alpha + h\sum_{k=1}^{N-1} f(x_k) + \beta = 0$, if not there is no discrete solution.
- we solve the system: (46) getting $(v_1, \ldots, v_N)^T$. The value v_0 can be assigned arbitrarily, the simplest is to take $v_0 = 0$
- The solution $\vec{u} = C\vec{v} = v_0\vec{1} + (0, v_1, \ldots, v_N)^T$

6.3 Finite Element Method (FEM) in 1D

Another discretization method for BVP - FInite Element Metho.

6.3.1 BVP in a variational formulation

We consider the same problem:

$$-u'' + c(x)u = f(x) \in (a, b) \tag{47}$$

and Dirichlet zero boundary conditions:

$$u(a) = u(b) = 0 \tag{48}$$

c nonnegative function Weak formulation: We reformulated the problem is so called weak or variational formulation, namely we take a function v such that v(a)=v(b)=0 and multiply the equation then integrate and use integration y parts formulas: find $u \in V$ s.t. (details in the script notes soon):

$$\int_a^b u'v' + cuvdx = \int_a^b fvdx \quad \forall v \in V. \tag{49}$$

V is a sufficiently large space of functions which are zero at a and b (e.g. think about piecewise C^1 functions) Mesh: Then to get a discrete problem we introduce: mesh (not necessarily equidistant):

$$a = x_0 < x_1 < ... < x_N = b, \quad h = \max_k |x_k - x_{k+1}| \tag{50}$$

6.3.2 Discrete linear FEM space

We introduce a linear discrete FEM space:

$$V_h = \{u \in C([a,b]) : u \text{ linear on } [x_k, x_{k+1}, u(a) = u(b) = 0\} \tag{51}$$

and replace the original weak formulation of BVP with the discrete one: Discrete FEM problem: find $u_h \in V_h$ s.t.

$$\int_a^b u_h' v' + c u_h v dx = \int_a^b f v dx \quad \forall v \in V_h \tag{52}$$

i.e. we replace the original functional space V by V_h (which in the simplest case is a subspace of V) and naturally we get a different discrete solution u_h. Nodal basis of V_h: In our case we introduce a natural so called nodal basis:

$$(g_k)_{k=1}^{N-1} \tag{53}$$

of functions in V_h s.t.

$$g_k(x_l) = 1 \quad k = l, \quad \text{or} \quad g_k(x_l) = 0 \quad k \neq l \tag{54}$$

Then we can represent $u_h = \sum_k y_k g_k$ and if we substitute it into the discrete variational equation and g_l for the test function v we get:

$$\int_a^b \sum_k (y_k g_k' g_l' + c g_k g_l) dx = \int_a^b f g_l dx \quad \forall\, l = 1, \ldots, N-1 \tag{55}$$

getting a linear systems

$$A\vec{y} = \vec{F} \tag{56}$$

with a symmetric, tridiagonal (why?), positive definite matrix

$$A = (\int_a^b g_l' g_k' + c(x) g_l g_k dx)_{l,k=1}^{N-1} \tag{57}$$

(Problem - compute its nonzero entries), sough after vector of entries of u_h, the rhs vector:

$$\vec{F} = (\int_a^b f g_l dx)_{l=1}^{N-1}$$

- naturally the integrals are computed in an approximated numerical way... Note that (problem) in case of equidistant points: $x_k = a + k * h$ with $h = (b-a)/N$ we get that A is tridiagonal with constant diagonals: $2/h$ on the main diagonal and $-1/h$ on super- and -sub diagonals.. i.e we get the same matrix only scaled by h if we compare our FEM system with the standard FDM one.. (3 point stencil)

6.3.3 Right hand side of the system - integration

How to approximate integrals

$$\int_a^b f g_k dx$$

- the simplest is to take a quadrature rule which use only the nodal points: $(x_k)_k$ i.e. the trapezoidal rule seems the most reasonable. The trapezoidal quadrature on $[c, d]$:

$$\int_c^d f dx \approx 0.5(d-c)(f(c)+f(d)) \tag{58}$$

The trapezoidal rule for mesh $a = x_0 < \ldots < x_N = b$: we apply the trapezoidal rule on each subinterval: $[x_{k-1}, x_k]$:

$$\int_a^b g dx \approx \sum_{k=1}^{N} 0.5(g(x_{k-1})+g(x_k))(x_k - x_{k-1}) \tag{59}$$

In our case we have that: $(f g_k)(x_l)$ is nonzero only for $l = k$:

$$\int_a^b f g_k dx = \int_{x_{k-1}}^{x_{k+1}} f g_k dx \approx 0.5 f(x_k) * |x_{k+1} - x_{k-1}| \tag{60}$$

if we have equaidistant mesh: $|x_{k+1} - x_{k-1}| = 2 * h$ and we get that

$$\int_a^b f g_k dx \approx f(x_k) * h \tag{61}$$

i.e. we get that the linear FEM system for equidistant mesh with the RHS vector approximated by the trapezoidal rule is the same as the system obtained by the FDM 3point stencil approximation of the same problem, anyway FEM is much more flexible and with well developed convergence theory, FDM can be usually considered as a special version of FEM (after using quadratures etc)

6.3.4 Neumann boundary condition at the right end

We consider another type of mixed boundary condtions, namely, a boundary value problem:

$$-u'' + c(x)u = f(x) \ \in (a, b) \tag{62}$$

and left Dirichlet zero boundary condition:

$$u(a) = 0 \tag{63}$$

and a right Neumann one:

$$u'(b) = \beta \tag{64}$$

We then consider a new space V_l with functions being zero at the left end (Dirichlet bnd condition - in general in FEM- Dirichlet conditions go into the definition of the space) and the weak formulation (again integrating by parts):

$$u \in V_l \quad \int_a^b u'v' + cuvdx = \int_a^b fvdx + u'(b)v(b) \quad \forall v \in V_l \tag{65}$$

New linear FEM space (functions that are zero at the left end only):

$$V_{h,l} = \{u \in C([a,b]) : u(a) = 0, u \text{ linear on } [x_k, x_{k+1}]\} \tag{66}$$

and variational problem:

$$u \in V_{h,l} \qquad \int_a^b u_h'v' + cu_hvdx = \int_a^b fvdx + \beta * v(b) \quad \forall v \in V_{h,l} \tag{67}$$

Nodal basis $(g_k)_{k=1,\ldots,N}$ (one more extra nodal function related to the end b.) and we get the system

$$A_l * \vec{U} = \vec{F_l} \tag{68}$$

A_l has an extra row and column (but is still symmetric and positive definite and tridiagonal - problem) F_l has also an extra entry i.e $\vec{F_l}^T = (\vec{F}^T; F_l(N))$ the last one namely:

$$F_l(N) = \int_a^b fg_Ndx + \beta * g_N(b)$$

which can be approximated (integral by trapezoidal rule on $[x_{N-1}, x_N]$) as

$$0.5f(b)(x_N - x_{N-1}) + \beta$$

in case of equidistant mesh:

$$0.5 * f(b)h + \beta$$

- we get that this equation (the whole one not just the rhs..) is equivalent to the FDM approximation equation of Neumann condition by the backward difference + correction by utilizing the ODE equation at b. Thus in FEM we get straightforwardly a good approximation of the Neumann type conditions.

6.3.5 Other classical FEM spaces

We can get a higher order approximations of both problems by just exchanging discrete spaces, e.g. for zero Dirichlet boundary conditions we can take a space of continuous

piecewise quadratic functions which are zero at the ends getting classical quadratic FEM space and then method... The unknowns are usually taken as the values at the nodes and at the midpoints. The order of convergence is higher than the one of the linear FEM in an appropriate norm if the solution is smooth enough. Naturally we have to use a quadrature rule which utilizes all respective nodal points, i.e. the nodes x_k and midpoints $0.5(x_k + x_{k+1})$. The most natural seems the composite Simpson rule.