Numerical Methods Library for OCTAVE

USER'S GUIDE

Lilian Calvet

November 13, 2008

Contents

1 How to install and use NMLibforOctave

In this part we explain how to install and to use the NMLibforOctave :

1.1 On Windows

Once the installation of Octave (3.0.0 and upper version) is finished you have to add a new path corresponding to the NMLibforOctave directory :

- Place the directory NMLibforOctave on the installation directory ~/Octave/ .
- Open ~ /.octaverc .
- Add the line : 'addpath("~/Octave/NMLibforOctave");'.
  If there is a space ' ' in a name of a subdirectory, add the character '\' before.
  Example : "C:/Program Files/Octave/NMLibforOctave"
  → addpath("C:/Program\ Files/Octave/NMLibforOctave");
- Save and exit.

1.2 On Linux

Once the installation of Octave (3.0.0 and upper version) is finished you have to add a new path corresponding to the NMLibforOctave directory :

- You can place the directory NMLibforOctave on the installation directory.
  For example on Ubuntu after installing octave with your package manager (package name : octave 3.0) copy NMLibforOctave in /usr/share/octave/3.0.0/m').
- Open ~ /.octaverc .
- Add the line : 'addpath("~/NMLibforOctave");'.
- Save and exit.

2 About the NMLibforOctave's content

The library NMLibforOctave's content can be decomposed is different application fields :

1. Linear systems
   - Gauss elimination
   - LU factorization
   - Cholesky factorization
   - Jacobi
   - Gauss-Seidel
   - Conjugate Gradient

- MINRES
- GMRES

2. Linear least squares
    - Normal equation
    - Householder
    - SVD
3. Nonlinear equations
    - Bisection
    - Fixed-point
    - Newton-Raphson
    - Secant
    - Newton's method for systems of nonlinear equations
4. Interpolation
    - Monomial basis
    - Lagrange interpolation
    - Newton interpolation
5. Numerical integration
    - Trapezoid's rule
    - Simpson's rule
    - Newton-Cotes' rule
6. Eigenvalue problems
    - Power iteration
    - Inverse method
    - Rayleigh quotient iteration
    - Orthogonal iteration
    - QR iteration
7. Optimization
    - Newton's method
    - Conjugate gradient method
    - Lagrange multipliers
8. Initial value problems for Ordinary differential Equations

- Euler
- Implicit Euler
- Modified Euler
- Fourth-order Rounge-Kutta
- Fourth-order predictor

9. Boundary value problems for Ordinary differential Equations
    - Shooting method
    - Finite difference method
    - Colocation method
10. Partial Differential Equations
    - Method of lines (for Heat equation)
    - Finite difference method for time-dependant PDEs (2-D solver for Advection, Heat and Wave equations) :
        - explicit method for Advection equation
        - implicit method for Advection equation
        - explicit method for Heat equation
        - implicit method for Heat equation
        - explicit method for Wave equation
        - implicit method for Wave equation
    - Finite difference method for time-independant PDEs (2-D solver for the Poisson equation)

3 Functions' description and use

3.1 Linear systems

Systems of linear algebraic equations arise in almost every aspect of applied mathematics and scientific computation. Such systems often occur naturally, but they are also frequently the result of approximating nonlinear equations by linear equations or differential equations by algebraic equations. For these reasons, the efficient and accurate solution of linear systems forms the cornerstone of many numerical methods for solving a wide variety of practical computational problems.

In matrix-vector notation, a system of linear algebraic equations has the form

$$Ax = b \tag{1}$$

where A is an m×n matrix, b is a given m-vector, and x is the unknown solution n-vector to be determined. There may or may not be a solution; and if there is a solution, it may or may not be unique. In this section we will consider only square systems solver, which means that m = n, i.e., the matrix has the same number of rows and columns.

3.1.1 Jacobi

[X, RES, NBIT] = JACOBI(A,B,X0,ITMAX,TOL) computes the solution of the linear system A*X = B with the Jacobi's method. If JACOBI fails to converge after the maximum number of iterations or halts for any reason, a message is displayed.

Parameters
A a square matrix.
B right hand side vector.
X0 initial point.
ITMAX maximum number of iteration.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
RES norm of the residual in X solution.
NBIT number of iterations to compute X solution.

3.1.2 Gauss-Seidel

[X, RES, NBIT] = GAUSS_SEIDEL(A,B,X0,ITMAX,TOL) computes the solution of the linear system A*X = B with the Gauss-Seidel's method. If GAUSS_SEIDEL fails to converge after the maximum number of iterations or halts for any reason, a message is displayed.

Parameters
A a square matrix.
B right hand side vector.
X0 initial point.
ITMAX maximum number of iteration.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
RES norm of the residual in X solution.
NBIT number of iterations to compute X solution.

3.1.3 MINRES

[X,FLAG,RELRES,ITN,RESVEC] = MINRES(A,B,RTOL,MAXIT) solves the linear system of equations A*X = B by means MINRES iterative method.

Parameters
A square (preferably sparse) matrix. In principle A should be symmetric.
B right hand side vector.
TOL relative tolerance for the residual error.
MAXIT maximum allowable number of iterations.

```
X0 initial guess.

Returns
X computed approximation to the solution.
RELRES ratio of the final residual to its initial value, measured in the
Euclidean norm.
ITN actual number of iterations performed.
RESVEC describes the convergence history of the method.
```

3.1.4 GMRES

```
[X, NBIT, BCK_ER, FLAG] = GMRES(A,B,X0,ITMAX,M,TOL,M1,M2,LOCATION) attempts
to solve the system of linear equations A*x = b for x.  The n-by-n coefficient
matrix A must be square and should be large and sparse.  The column vector
B must have length n.  If GMRES fails to converge after the maximum number
of iterations or halts for any reason, a message is displayed.  GMRES restarts
all m-iterations.

Parameters
A a square matrix.
B right hand side vector.
X0 initial guess.
ITMAX maximum number of iteration.
M GMRES restarts all m-iterations.
TOL tolerance on the stopping criterion.
M1,M2 preconditionners
LOCATION preconditionners' location :  if left then location=1 else right
and location=2.

Returns
X computed solution.
NBIT number of iterations to compute X solution.
BCK_ER describes the convergence history of the method.
FLAG if the method converges then FLAG=0 else FLAG=-1.
```

3.1.5 Already existing functions about linear solver

It already exists function to solve linear systems in Octave. We have particularly the Conjugate Gradient method *pcg*, the Cholesky factorization *chol* and finally LU factorization *lu*. To see how to use these function use the command 'help <function_name>' in Octave.

3.1.6 What is hidden behind the command '\'

It is useful to know that the specific algorithm used by Octave when the command is invoked depends upon the structure of the matrix A. For a system with dense matrix, Octave only uses the LU or the QR factorization. When the matrix is sparse Octave follows this procedure :

1. if the matrix is upper (with column permutations) or lower (with row permutations) triangular, perform a sparse forward or backward substitution;
2. if the matrix is square, symmetric with a positive diagonal, attempt sparse Cholesky factorization;
3. if the sparse Cholesky factorization failed or the matrix is not sym- metric with a positive diagonal, factorize using the UMFPACK li- brary;
4. if the matrix is square, banded and if the band density is "small enough" continue, else goto 3;
   (a) if the matrix is tridiagonal and the right-hand side is not sparse continue, else goto b);
      i. if the matrix is symmetric, with a positive diagonal, attempt Cholesky factorization;
      ii. if the above failed or the matrix is not symmetric with a positive diagonal use Gaussian elimination with pivoting;
   (b) if the matrix is symmetric with a positive diagonal, attempt Cholesky factorization;
   (c) if the above failed or the matrix is not symmetric with a positive diagonal use Gaussian elimination with pivoting;
5. if the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, find a solution in the least-squares sense.

◈Among above-cited solvers, GMRES, MINRES and Conjugate Gradient are used to solve "large" linear system, i.e. , with "large" n.

1. **MINRES** is used to solve linear systems with **symmetric indefinite matrices**.
2. **Conjugate Gradient Method** is used with **symmetric positive definite matrix**.
3. **GMRES** is used in other cases.

Notice that there are no pre-established rules to solve a linear system. It depends on the size, on the sparse structure... of the system. Kind and location of preconditioners can also be very important.

3.2 Linear least squares

What meaning should we attribute to a system of linear equations Ax = b if the matrix A is not square? Since a nonsquare matrix cannot have an inverse, the system of equations must have either no solution or a nonunique solution. Nevertheless, it is often useful to define a unique vector x that satisfies the linear system in an approximate sense. In this section we will consider methods for solving such problems.

Let A be an m ×n matrix. We will be concerned with the most commonly occurring case, m > n, which is called overdetermined because there are more equations than unknowns.

3.2.1 Normal equation

```
[X, RES] = NORMALEQ(A,B) computes the solution of linear least squares problem
min(norm(A*X-B,2)) solving associated normal equation A'*A*X = A'*B.

Parameters
A a matrix.
B right hand side vector.
Returns

X computed solution.
RES value of norm(A*X-B) with X solution computed.
```

3.2.2 Householder

```
[X, RES] = LLS_HQR(A,B) computes the solution of linear least squares problem
min(norm(A*X-B,2)) using Householder QR.

Parameters
A a matrix.
B right hand side vector.

Returns
X computed solution.
RES value of norm(A*X-B) with X solution computed.
```

3.2.3 SVD

```
[X, RES] = SVD_LEAST_SQUARES(A,B) computes the solution of linear least squares
problem min(norm(A*X-B,2)) using the singular value decomposition of A.

Parameters
A a matrix.
B right hand side vector.

Returns
X computed solution.
RES value of norm(A*X-B) with X solution computed.
```

3.3 Nonlinear equations

We will now consider methods for solving nonlinear equations. Given a nonlinear function f , we seek a value x for which

$$f(x) = 0. \tag{2}$$

Such a solution value for x is called a root of the equation, and a zero of the function f . Though technically they have distinct meanings, these two terms are informally used more or

less interchangeably, with the obvious meaning. Thus, this problem is often referred to as root finding or zero finding. In discussing numerical methods for solving nonlinear equations, we will distinguish two cases:

$$f : \mathbb{R} \to \mathbb{R} \text{ (scalar)}, \tag{3}$$

and

$$f : \mathbb{R}^n \to \mathbb{R}^n \text{ (vector)}. \tag{4}$$

The latter is referred to as a system of nonlinear equations, in which we seek a vector x such that all the component functions of f (x) are zero *simultaneously*.

3.3.1 Bisection

```
X = BISECTION(FUN,A,B,ITMAX,TOL) tries to find a zero X of the continuous
function FUN in the interval [A, B] using the bisection method.  FUN accepts
real scalar input x and returns a real scalar value.  If the search fails
an error message is displayed.  FUN can also be an inline object.

[X, RES, NBIT] = BISECTION(FUN,A,B,ITMAX,TOL) returns the value of the residual
in X solution and the iteration number at which the solution was computed.

Parameters
FUN evaluated function.
A,B [A,B] interval where the solution is computed, A < B and sign(FUN(A))
= - sign(FUN(B)).
ITMAX maximal number of iterations.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
NBIT number of iterations to find the solution.
RES value of the residual in X solution.
```

3.3.2 Fixed-point

```
X = FIXED_POINT(FUN,X0,ITMAX,TOL) solves the scalar nonlinear equation such
that 'FUN(X) == X' with FUN continuous function.  FUN accepts real scalar
input X and returns a real scalar value.  If the search fails an error message
is displayed.  FUN can also be inline objects.

[X, RES, NBIT] = FIXED_POINT(FUN,X0,ITMAX,TOL) returns the norm of the residual
in X solution and the iteration number at which the solution was computed.

Parameters
FUN evaluated function.
DFUN f's derivate.
```

X0 initial point.
ITMAX maximal number of iteration.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
RES norm of the residual FUN(X)-X in X solution.
NBIT number of iterations to find the solution.

3.3.3 Newton-Raphson

X = NLE_NEWTRAPH(FUN,DFUN,X0,ITMAX,TOL) tries to find a zero X of the continuous and differentiable function FUN nearest to X0 using the Newton-Raphson method. FUN and its derivate DFUN accept real scalar input x and returns a real scalar value. If the search fails an error message is displayed. FUN and DFUN can also be inline objects.

[X, RES, NBIT] = NLE_NEWTRAPH(FUN,DFUN,X0,ITMAX,TOL) returns the value of the residual in X solution and the iteration number at which the solution was computed.

Parameters
FUN evaluated function.
DFUN f's derivate.
X0 initial point.
ITMAX maximal number of iterations.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
RES value of the residual in x solution.
NBIT number of iterations to find the solution.

3.3.4 Secant

X = SECANT(FUN,X1,X2,ITMAX,TOL) tries to find a zero X of the continuous function FUN using the secant method with starting points X1, X2. FUN accepts real scalar input X and returns a real scalar value. If the search fails an error message is displayed. FUN can also be an inline object.

[X, RES, NBIT] = SECANT(FUN,X1,X2,ITMAX,TOL) returns the value of the residual in X solution and the iteration number at which the solution was computed.

Parameters
FUN evaluated function.
X1,X2 starting points.

```
ITMAX maximal number of iterations.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
RES value of the residual in X solution.
NBIT number of iterations to find the solution.
```

3.3.5 Newton's method for systems of nonlinear equations

```
X = NLE_NEWTSYS(FFUN,JFUN,X0,ITMAX,TOL) tries to find the vector X, zero
of a nonlinear system defined in FFUN with jacobian matrix defined in the
function JFUN, nearest to the vector X0.

[X, RES, NBIT] = NLE_NEWTSYS(FUN,DFUN,X0,ITMAX,TOL) returns the norm of the
residual in X solution and the iteration number at which the solution was
computed.

Parameters
FFUN evaluated function.
JFUN FFUN's jacobian matrix.
X0 initial point.
ITMAX maximal number of iterations.
TOL tolerance on the stopping criterion.

Returns
X computed solution.
RES norm of the residual in X solution.
NBIT number of iterations to find the solution.
```

3.4 Interpolation

Interpolation simply means fitting some function to given data so that the function has the same values as the given data. In general, the simplest interpolation problem in one dimension is of the following form: for given data

$$(t_i, y_i), \quad i = 1, ..., n, \tag{5}$$

with $t_1 < t_2 < \ldots < t_n$, we seek a function f such that

$$f(t_i) = y_i, \quad i = 1, ..., n. \tag{6}$$

We call f an interpolating function, or simply an interpolant, for the given data. It is often desirable for f (t) to have "reasonable" values for t between the data points, but such a requirement may be difficult to quantify. In more complicated interpolation problems, additional data might be prescribed, such as the slope of the interpolant at given points, or additional constraints might be imposed on the interpolant, such as monotonicity, convexity, or the degree of smoothness required.

3.4.1 Monomial basis

```
P = ITPOL_MONOM(X,Y,x) computes the monomial basis interpolation of points
defined by x-coordinate X and y-coordinate Y. x can be a real vector, each
row in the solution array P corresponds to a x-coordinate in the vector x.

Parameters
X abscissas of interpolated points.
Y odinates of interpolated points.
x can be a scalar or a vector of values.

Returns
P value of p(x).
```

3.4.2 Lagrange interpolation

```
P = LAGRANGE(X,Y,x) computes the polynomial Lagrange interpolation of points
defined by x-coordinate X and y-coordinate Y. x can be a real vector, each
row in the solution array P corresponds to a x-coordinate in the vector x.

Parameters
X abscissas of interpolated points.
Y odinates of interpolated points.
x can be a scalar or a vector of values.

Returns
P value of P(x).
```

3.4.3 Newton interpolation

```
P = ITPOL_NEWT(X,Y,x) computes the polynomial Newton interpolation of points
defined by x-coordinate X and y-coordinate Y. x can be a real vector, each
row in the solution array P corresponds to a x-coordinate in the vector x.

Parameters
X abscissas of interpolated points.
Y odinates of interpolated points.
x can be a scalar or a vector of values.

Returns
P value of p(x).
```

3.5 Numerical integration

The numerical approximation of definite integrals is known as numerical quadrature. This name derives from ancient methods for computing areas of curved figures, the most famous example of which is the problem of "squaring the circle" (finding a square having the same area as a

given circle). In our case we wish to compute the area under a curve defined over an interval on the real line. Thus, the quantity we wish to compute is of the form

$$I(f) = \int_a^b f(x)dx. \tag{7}$$

We will generally take the interval of integration to be finite, and we will assume for the most part that the integrand f is continuous and smooth. We will consider only briefly how to deal with an infinite interval of integration or an integrand function that may have discontinuities or singularities.

Note that we seek a single number as an answer, not a function or a symbolic formula. This feature distinguishes numerical quadrature from the solution of differential equations or the evaluation of indefinite integrals, as in elementary calculus and in many packages for symbolic computation.

An integral is, in effect, an infinite summation. It should come as no surprise that we will approximate this infinite sum by a finite sum. Such a finite sum, in which the integrand function is sampled at a finite number of points in the interval of integration, is called a quadrature rule. Our main object of study will be how to choose the sample points and how to weight their contributions to the quadrature formula so that we obtain a desired level of accuracy at a reasonable computational cost. For numerical quadrature, computational work is usually measured by the number of evaluations of the integrand function that are required.

3.5.1 Trapezoid's rule

```
RES = INTE_TRAPEZ(FUN,A,B,N) computes an approximation of the integral of
the function FUN via the trapezoid method (using N equispaced intervals).
FUN accepts real scalar input x and returns a real scalar value.  FUN can
also be an inline object.

Parameters
FUN integrated function.
A,B FUN is integrated on [A,B].
N number of subdivisions.

Returns
RES result of integration.
```

3.5.2 Simpson's rule

```
RES = INTE_SIMPSON(FUN,A,B,N) computes an approximation of the integral of
the function FUN via the Simpson method (using N equispaced intervals).  FUN
accepts real scalar input x and returns a real scalar value.  FUN can also
be an inline object.

Parameters
FUN integrated function.
A,B FUN is integrated on [A,B].
```

```
N number of subdivisions.

Returns
RES result of integration.
```

3.5.3 Newton-Cotes' rule

```
RES = INTE_NEWTCOT(FUN,A,B,N) computes an approximation of the integral of
the function FUN via the Newton-Cotes method (using N equispaced intervals).
FUN accepts real scalar input x and returns a real scalar value.  FUN can
also be an inline object.

Parameters FUN integrated function.  A,B FUN is integrated on [A,B]. N number
of subdivisions.

Returns RES result of integration.
```

3.6 Eigenvalue problems

The standard algebraic eigenvalue problem is as follows: Given an n ×n matrix A, find a scalar λ and a nonzero vector x such that

$$Ax = \lambda x. \tag{8}$$

Such a scalar λ is called an eigenvalue, and x is a corresponding eigenvector. The set of all the eigenvalues of a matrix A, denoted by λ(A), is called the spectrum of A.

An eigenvector of a matrix determines a direction in which the effect of the matrix is particularly simple: The matrix expands or shrinks any vector lying in that direction by a scalar multiple, and the expansion or contraction factor is given by the corresponding eigenvalue λ. Thus, eigenvalues and eigenvectors provide a means of understanding the complicated behavior of a general linear transformation by decomposing it into simpler actions.

Eigenvalue problems occur in many areas of science and engineering. For example, the natural modes and frequencies of vibration of a structure are determined by the eigenvectors and eigenvalues of an appropriate matrix. The stability of the structure is determined by the locations of the eigenvalues, and thus their computation is of critical interest. We will also see later in this book that eigenvalues can be very useful in analyzing numerical methods, such as the convergence analysis of iterative methods for solving systems of algebraic equations, and the stability analysis of methods for solving systems of differential equations.

3.6.1 Power iteration

```
[LAMBDA, V, NBIT] = EIG_POWER(A, X0, ITMAX, TOL) computes dominant eigenvalue
and associated eigenvector of A with power iteration method.  If EIG_POWER
fails to converge after the maximum number of iterations or halts for any
reason, a message is displayed.

Parameters
```

A a square matrix.
X0 initial point.
ITMAX maximal number of iterations.
TOL maximum relative error.

Returns
LAMBDA dominant eigenvalue of A.
V associated eigenvector.
NBIT number of iteration to the solution.

3.6.2 Inverse method

[LAMBDA, V, NBIT] = EIG_INVERSE(A, X0, ITMAX, TOL) Compute the smallest eigenvalue of A and associated eigenvector with inverse method. If EIG_INVERSE fails to converge after the maximum number of iterations or halts for any reason, a message is displayed.

Parameters
A a square matrix.
X0 initial point.
ITMAX maximal number of iterations.
TOL maximum relative error.

Returns
LAMBDA smallest eigenvalue of A.
V associated eigenvector.
NBIT number of iteration to the solution.

3.6.3 Rayleigh quotient iteration

[LAMBDA, V, NBIT] = EIG_RAYLEIGH(A, X0, ITMAX, TOL) computes the best estimate of an eigenvalue of A associated to an approximate eigenvector X0 with Rayleigh quotient iteration method. If EIG_RAYLEIGH fails to converge after the maximum number of iterations or halts for any reason, a message is displayed. This method can be used to accelerate the convergence of a method such as power iteration.

Parameters
A a square matrix.
X0 initial point corresponding to an approximate eigenvector.
ITMAX maximal number of iterations.
TOL maximum relative error.

Returns
LAMBDA the best estimate for the corresponding eigenvalue.
V associated eigenvector.
NBIT number of iteration to the solution.

3.6.4 Orthogonal iteration

```
[LAMBDA, V, NBIT] = EIG_ORTHO(A, X0, ITMAX, TOL) computes P=size(X0,2) eigenvalues
and associated eigenvectors of A with orthogonal iteration method.  If EIG_ORTHO
fails to converge after the maximum number of iterations or halts for any
reason, a message is displayed.

Parameters
A a square matrix.
X0 arbitrary N x P matrix of rank P, contains X0(1),X0(2),...X0(P)
linearly independant.
ITMAX maximal number of iterations.
TOL maximum relative error.

Returns
LAMBDA P-vector containing eigenvalues of A.
V P eigenvectors.
NBIT number of iteration to the solution.
```

3.6.5 QR iteration

```
[LAMBDA, V, NBIT] = EIG_QR(A, ITMAX, TOL) computes N (=size(A)) eigenvalues
and associated eigenvectors of A with orthogonal iteration method.  If EIG_QR
fails to converge after the maximum number of iterations or halts for any
reason, a message is displayed.

Parameters
A (N*N) matrix
ITMAX maximal number of iterations.
TOL maximum relative error.

Returns
LAMBDA N-vector containing eigenvalues of A.
V N associated eigenvectors.
NBIT number of iteration to the solution.
```

3.7 Optimization

We now turn to the problem of determining extreme values, or optimum values (maxima or minima), that a given function has on a given domain. More formally, given a function $f : \mathbb{R}^n \to \mathbb{R}$, and a set $\mathcal{S} \subseteq \mathbb{R}^n$, we seek $x \in \mathcal{S}$ such that f attains a minimum on $\mathcal{S}$ at x, i.e., $f(x) \leq f(y)$ for all $y \in \mathcal{S}$. Such a point x is called a minimizer , or simply a minimum, of f . Since a maximum of f is a minimum of f , it suffices to consider only minimization. The objective function, f , may be linear or nonlinear, and it is usually assumed to be differentiable. The constraint set $\mathcal{S}$ is usually defined by a system of equations or inequalities, or both, that may be linear or nonlinear. A point $x \in \mathcal{S}$ that satisfies the constraints is called a feasible point. If $\mathcal{S} = \mathbb{R}^n$, then the problem is unconstrained . General continuous optimization problems have

the form

$$\min_x \quad f(x) \text{ subject to } \quad g(x) = 0 \text{ and } h(x) \leq 0, \tag{9}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, and $h : \mathbb{R}^n \to \mathbb{R}^k$. Optimization problems are classified by the properties of the functions involved. For example, if f , g, and h are all linear, then we have a linear programming problem. If any of the functions involved are nonlinear, then we have a nonlinear programming problem. Important subclasses of the latter include problems with a nonlinear objective function and linear constraints, or a nonlinear objective function and no constraints.

Newton's method and Conjugate Gradient's method are directly used in unconstrained optimization and Lagrange multipliers are used in constrained optimization.

3.7.1 Newton's method

```
[X, FX, NBIT] = OPT_NEWTON(FUN, GFUN, HFUN, X0, ITMAX, TOL) computed the
minimum of the FUN function with the newton method nearest X0.  Function
GFUN defines the gradient vector and function HFUN defines the hessian matrix.
FUN accepts a real vector input and return a real vector.  FUN, GFUN and
HFUN can also be inline object.  If OPT_NEWTON fails to converge after the
maximum number of iterations or halts for any reason, a message is displayed.

Parameters
FUN evaluated function.
GFUN FUN's gradient function.
HFUN FUN's hessian matrix function.
X0 initial point.
ITMAX maximal number of iterations.
TOL tolerance on the stopping criterion.

Returns
X computed solution of min(FUN).
FX value of FUN(X) with X computed solution.
NBIT number of iterations to find the solution.
```

◈The iteration scheme for Newton' s method has the form

$$x_{k+1} = x_k - H_f^{-1}(x_k) f(x_k), \tag{10}$$

where $H_f(x)$ is the Hessian matrix of second partial derivatives of f ,

$$\{H_f(x)\}_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}, \tag{11}$$

evaluated at x_k . As usual, we do not explicitly invert the Hessian matrix but instead use it to solve a linear system

$$H_f(xk)s_k = f(x_k) \tag{12}$$

for s_k , then take as next iterate

$$x_{k+1} = x_k + s_k. \tag{13}$$

The convergence rate of Newton's method for minimization is normally quadratic. As usual, however, Newton's method is unreliable unless started close enough to the solution.

3.7.2 Conjugate gradient method

```
[X, FX, NBIT] = OPT_CG(FUN, X0, GFUN, HFUN, TOL, ITMAX) computed the minimum
of the FUN function with the conjugate gradient method nearest X0.  Function
GFUN defines gradient vector and function HFUN defines hessian matrix.  FUN
accepts a real vector input and return a real vector.  FUN, GFUN and HFUN
can also be inline object.  If OPT_CG fails to converge after the maximum
number of iterations or halts for any reason, a message is displayed.

Parameters
FUN evaluated function.
X0 initial point.
GFUN FUN's gradient function.
HFUN FUN's hessian matrix function.
TOL tolerance on the stopping criterion.
ITMAX maximal number of iterations.

Returns
X computed solution of min(FUN).
FX value of FUN(X) with X computed solution.
NBIT number of iterations to find the solution.
```

◈The conjugate gradient method is another alternative to Newton's method that does not require explicit second derivatives. Indeed, unlike secant updating methods, the conjugate gradient method does not even store an approximation to the Hessian matrix, which makes it especially suitable for very large problems.
The conjugate gradient method uses gradients, but it avoids repeated searches by modifying the gradient at each step to remove components in previous directions. The resulting sequence of conjugate (i.e., orthogonal in some inner product) search directions implicitly accumulates information about the Hessian matrix as iterations proceed. Theoretically, the method is exact after at most n iterations for a quadratic objective function in n dimensions, but it is usually quite effective for more general unconstrained minimization problems as well.

3.7.3 Lagrange multipliers

```
[XMIN, LAMBDAMIN, FMIN] = OPT_LAGRANGE(F, GRADF, G, JACG, X0) computed the
minimum of the function FUN subject to 'G(X) = 0' with the lagrange multiplier
method.  Function GRADF defines the gradient vector of F. Function G represents
equality-constrained and function JACG defines its jacobian matrix.  F and
G accept a real vector input and return a real vector.  F, GRADF, G and JACG
can also be inline object.

Parameters
F evaluated function.
GRADF F's gradient function.
G equality-constrained function :  'G(X) = 0'.
X0 initial point.

Returns
XMIN computed solution of min(FUN).
LAMBDAMIN vector of Lagrange multipliers on XMIN.
FX value of FUN(X) with X computed solution.
```

◈Consider the minimization of a nonlinear function subject to nonlinear equality constraints,

$$\min_x \quad f(x) \text{ subject to } \quad g(x) = 0, \tag{14}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$, with $m \leq n$.
The *Lagrangian function*, $\mathcal{L} : \mathbb{R}^{n+m} \to \mathbb{R}$, is given by

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T g(x), \tag{15}$$

whose gradient and Hessian are given by

$$\nabla \mathcal{L}(x, \lambda) = \begin{bmatrix} \mathcal{L}_x(x, \lambda) \\ \mathcal{L}_\lambda(x, \lambda) \end{bmatrix} = \begin{bmatrix} \nabla f(x) + J_g^T(x) \\ g(x) \end{bmatrix} \tag{16}$$

where J_g is the Jacobian matrix of g and λ is an m-vector of Lagrange multipliers.
Together, the necessary condition and the requirement of feasibility say that we are looking for a critical point of the Lagrangian function, which is expressed by the system of nonlinear equations

$$\begin{bmatrix} \nabla f(x) + J_g^T(x) \\ g(x) \end{bmatrix} = 0 \tag{17}$$

3.8 Initial value problems for Ordinary differential Equations

We determine a unique solution to the ODE $y'(t) = f(y, t)$ with $y(t_0) = t_0$, provided that f is continuously differentiable. Because the independent variable t usually represents time, we think of t_0 as the initial time and y_0 as the initial value. Hence, this is termed an initial value problem. The ODE governs the dynamic evolution of the system in time from its initial state y_0 at time t_0 onward, and we seek a function $y(t)$ that describes the state of the system as a function of time.

3.8.1 Euler

```
[TT,Y] = ODE_EULER(ODEFUN,TSPAN,Y,NH) with TSPAN = [T0, TF] integrates the
system of differential equations Y'=f(T,Y) from time T0 to TF with initial
condition Y0 using the forward Euler method on an equispaced grid of NH intervals.
Function ODEFUN(T,Y) must return a column vector corresponding to f(T, Y).
Each row in the solution array Y corresponds to a time returned in the column
vector T.

Parameters
ODEFUN integrated function.
TSPAN TSPAN = [T0 TF]
Y initial value Y(T0).
NH TT equispaced grid of NH intervals.

Returns
TT equispaced grid of NH intervals.
Y solution array.
```

3.8.2 Implicit Euler

```
[TT,Y] = ODE_BEULER(ODEFUN,TSPAN,Y,NH) with TSPAN = [T0, TF] integrates the
system of differential equations Y'=f(T,Y) from time T0 to TF with initial
condition Y0 using the backward Euler method on an equispaced grid of NH
intervals.  Function ODEFUN(T, Y) must return a column vector corresponding
to f(T, Y). Each row in the solution array Y corresponds to a time returned
in the column vector T.

Parameters
ODEFUN integrated function.
TSPAN TSPAN = [T0 TF].
Y initial value Y(T0).
NH TT equispaced grid of NH intervals.

Returns
TT equispaced grid of NH intervals.
Y solution array.
```

◈Euler's method bases its projection on the derivative at the current point, and the resulting large value causes the numerical solution to diverge radically from the desired solution. This behavior should not surprise us. The Jacobian for this equation is $J = 100$, so the stability condition for Euler's method requires a stepsize h < 0.02, which we are violating.
By contrast, the backward Euler method has no trouble solving this problem. In fact, the backward Euler solution is extremely insensitive to the initial value.

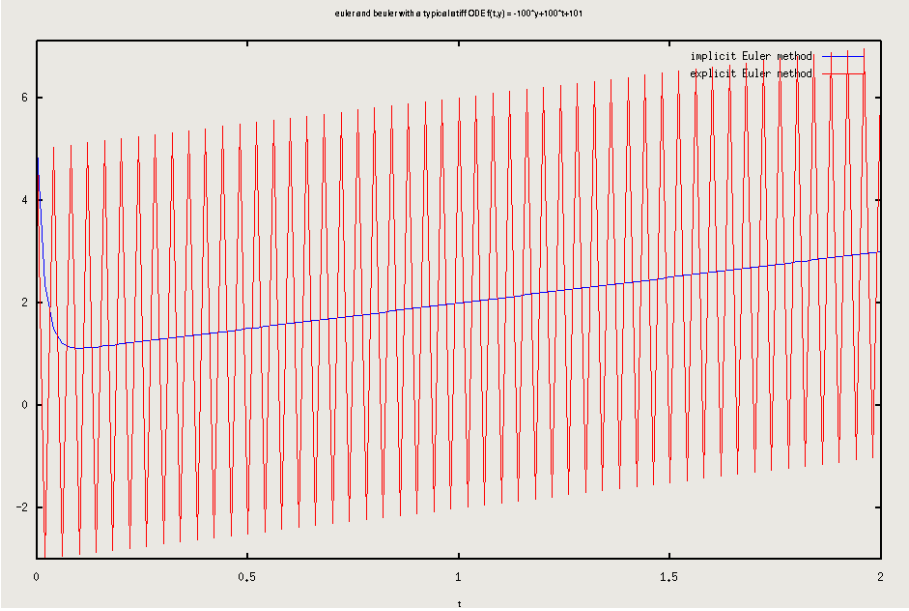

Figure 1: euler vs. beuler with a typical stiff ODE $y' = -100y + 100t + 101$

3.8.3 Modified Euler

```
[TT,Y] = ODE_EULER(ODEFUN,TSPAN,Y,NH) with TSPAN = [T0, TF] integrates the
system of differential equations Y'=f(T,Y) from time T0 to TF with initial
condition Y0 using the modified Euler method on an equispaced grid of NH
intervals.  Function ODEFUN(T,Y) must return a column vector corresponding
to f(T, Y). Each row in the solution array Y corresponds to a time returned
in the column vector T.

Parameters
ODEFUN integrated function.
TSPAN TSPAN = [T0 TF].
Y initial value Y(T0).
NH TT equispaced grid of NH intervals.

Returns
TT equispaced grid of NH intervals.
Y solution array.
```

3.8.4 Fourth-order Rounge-Kutta

It corresponds to ode23,ode45 which already exit in Octave. Write 'help <function_name>' in Octave to have more informations.

3.8.5 Fourth-order predictor

```
[TT,Y] = ODE_FOP(ODEFUN,TSPAN,Y,NH) with TSPAN = [T0, TF] integrates the
system of differential equations Y'=f(T,Y) from time T0 to TF with initial
```

```
condition Y0 using the fourth-order predictor scheme on an equispaced grid
of NH intervals.  Function ODEFUN(T,Y) must return a column vector corresponding
to f(T, Y). Each row in the solution array Y corresponds to a time returned
in the column vector T.

Parameters
ODEFUN integrated function.
TSPAN tspan = [T0 TF].
Y initial value Y(T0).
NH TT equispaced grid of NH intervals.

Returns
TT equispaced grid of NH intervals.
Y solution array.
```

3.9 Boundary value problems for Ordinary differential Equations

A boundary value problem for a differential equation specifies more than one point at which the solution or its derivatives must have given values. For example, a two-point boundary value problem for a second-order ODE has the form

$$y = f(t, y, y), \quad a \le t \le b, \tag{18}$$

with boundary conditions

$$y(a) = \alpha, \quad y(b) = \beta. \tag{19}$$

An initial value problem for such a second-order equation would have specified both y and y' at a single point, say, t_0 . These initial data would have supplied all the information necessary to begin a numerical solution method at t_0 , stepping forward to advance the solution in time (or whatever the independent variable might be).

3.9.1 Shooting method

```
[T,Y] = ODE_SHOOT(IVP, A, B, UA, UB) integrates the system of differential
equations u"= f(t,u,u') from time A to B with boundary conditions u(A) =
UA and u(B) = UB. Function IVP(t,u,u') must return a double column vector
[u', u"] with u"= f(t,u,u').  Each row in the solution array Y corresponds
to a time returned in the column vector T.

Parameters
IVP integrated function.
A T0.
B TF.
UA initial value Y(T0).
UB final value Y(TF).
```

```
Returns
T equispaced grid.
Y solution array.
```

◈The basic idea of the shooting method is illustrated in Fig. 2. Each curve represents a solution of the same second-order ODE, with different values for the initial slope giving different solution curves. All of the solutions start with the given initial value $y(a) = \alpha$, but for only one value of the initial slope does the resulting solution curve hit the desired boundary condition $y(b) = \beta$.

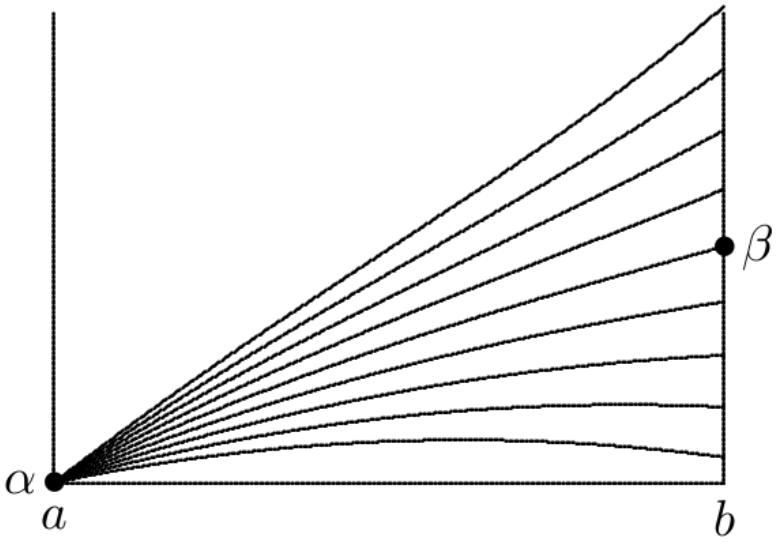

Figure 2: Shooting method for a two-point boundary value problem.

3.9.2 Finite difference method

```
[T,Y] = ODE_FINIT_DIFF(RHS, A, B, UA, UB, N) integrates the system of differential
equations u"= f(t,u,u') from time A to B with boundary conditions u(A) =
UA and u(B) = UB on an equispaced grid of N intervals.  Function RHS(t,u,u')
must return a column vector corresponding to f(t,u,u').  Each row in the
solution array Y corresponds to a time returned in the column vector T.

Parameters
RHS integrated function.
A T0.
B TF.
UA initial value Y(T0).
UB final value Y(TF).
N T equispaced grid of N intervals.

Returns
T equispaced grid of N intervals.
Y solution array.
```

3.9.3 Colocation method

```
[T,Y] = ODE_COLLOC(RHS, A, B, UA, UB, DN, N) integrates the system of differential
equations u"= f(t,u,u') from time A to B with boundary conditions u(A) =
UA and u(B) = UB on an equispaced grid of N intervals.  Function RHS(t,u,u')
must return a column vector corresponding to f(t,u,u').  Each row in the
solution array Y corresponds to a time returned in the column vector T.
```

```
Parameters
RHS integrated function.
A T0.
B TF.
UA initial value Y(T0).
UB final value Y(TF).
N T equispaced grid of N intervals.
DN degree of computed polynomial solution.

Returns
T equispaced grid of N intervals.
Y solution array.
```

3.10 Partial Differential Equations

We turn now to partial differential equations (PDEs), where many of the numerical tech- niques we saw for ODEs, both initial and boundary value problems, are also applicable. The situation is more complicated with PDEs, however, because there are additional inde- pendent variables, typically one or more space dimensions and possibly a time dimension as well. Additional dimensions significantly increase computational complexity. Problem formulation also becomes more complex than for ODEs, as we can have a pure initial value problem, a pure boundary value problem, or a mixture of the two. Moreover, the equation and boundary data may be defined over an irregular domain in space.

First, we establish some notation. For simplicity, we will deal only with single PDEs (as opposed to systems of several PDEs) with only two independent variables (either two space variables, which we denote by x and y, or one space and one time variable, which we denote by x and t). In a more general setting, there could be any number of dimensions and any number of equations in a coupled system of PDEs. We denote by u the unknown solution function to be determined and its partial derivatives with respect to the independent variables by appropriate subscripts: $u_x = \partial u/\partial x, u_{xy} = \partial^2 u/\partial x \partial y$, etc.

3.10.1 Method of lines (for Heat equation)

```
[T, X, U] = PDE_HEAT_LINES(NX, NT, C, F) solves the heat equation D U/DT
= C D2 U/DX2 with the method of lines on [0,1]x[0,1].  Initial condition
is U(0,X) = F. C is a positive constant.  NX is the number of space integration
intervals and NT is the number of time-integration intervals.

Parameters
NX X equispaced grid of NX intervals.
NT T equispaced grid of NX intervals.
C positive constant.

Returns
T equispaced grid of NT intervals.
```

```
X equispaced grid of NX intervals.
U solution array.
```

◈Notice that we use beuler to integrate each ODE of this system. In fact if we are computing the solution of the heat equation for example. After the finite difference approximation we obtain the system $y' = Ay$ with A = tridiag(1,-2,1). The Jacobian matrix A of this system has eigenvalues between $4c/(\Delta x)2$ and 0, which makes the ODE very stiff as the spatial mesh size Δx becomes small. This stiffness, which is typical of ODEs derived from PDEs in this manner, must be taken into account in choosing an ODE method for solving the semidiscrete system.

3.10.2 2-D solver for Advection equation

Advection equation :

$$u_t + cu_x = 0 \tag{20}$$

```
[T, X, U] = PDE_ADVEC_EXP(N, DX, K, DT, C, F) solves the advection equation
D U/DT = -C D U/DX with the explicit method on [0,1]x[0,1].  Initial condition
is U(0,X) = F. C is a positive constant.  N is the number of space integration
intervals and K is the number of time-integration intervals.  DX is the size
of a space integration interval and DT is the size of time-integration intervals.

Parameters
NX number of space integration intervals.
DX size of a space integration interval.
NT number of time-integration intervals.
DT size of time-integration intervals.
C positive constant.
F initial condition U(0,X) = F(X).

Returns
T grid of NT intervals.
X grid of NX intervals.
U solution array.
   [T, X, U] = PDE_ADVEC_IMP(N, DX, K, DT, C, F) solves the advection equation
D U/DT = -C D U/DX with the implicit method on [0,1]x[0,1].  Initial condition
is U(0,X) = F. C is a positive constant.  N is the number of space integration
intervals and K is the number of time-integration intervals.  DX is the size
of a space integration interval and DT is the size of time-integration intervals.

Parameters
NX number of space integration intervals.
DX size of a space integration interval.
NT number of time-integration intervals.
DT size of time-integration intervals.
C positive constant.
```

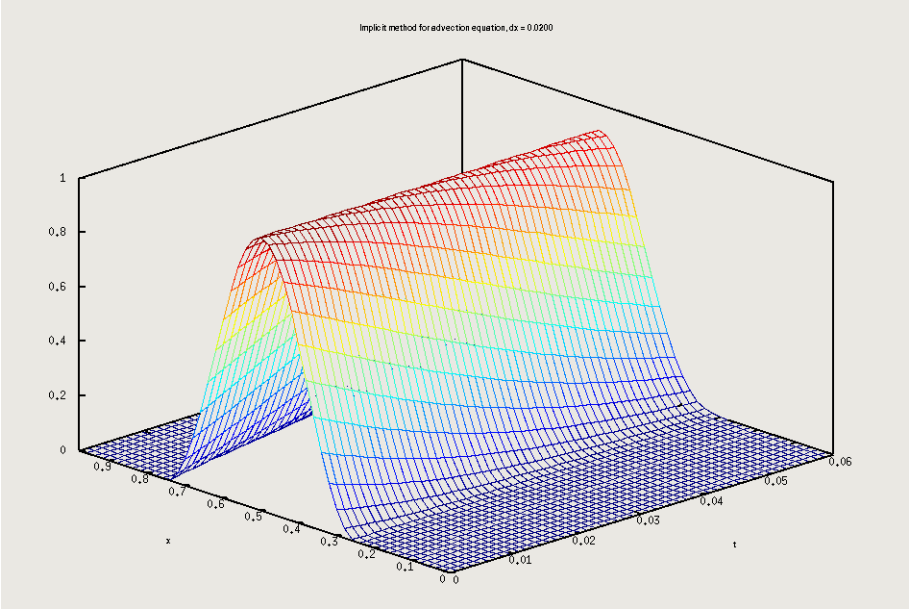

Figure 3: Solution of the Advection equation with $x \in [0, 1]$ and $t \in [0, 0.6]$.

```
F initial condition U(0,X) = F(X).

Returns
T grid of NT intervals.
X grid of NX intervals.
U solution array.
```

3.10.3 2-D solver for Heat equation

Heat equation :

$$u_t = cu_{xx} \tag{21}$$

```
   [T, X, U] = PDE_HEAT_EXP(N, DX, K, DT, C, F, ALPHA, BETA) solves the heat
equation D U/DT = C D2U/DX2 with the explicit method on [0,1]x[0,1].  Initial
condition is U(0,X) = F and boundary conditions are U(t,0) = ALPHA and U(t,1)
= beta.  C is a positive constant.  N is the number of space integration
intervals and K is the number of time-integration intervals.  DX is the size
of a space integration interval and DT is the size of time-integration intervals.

Parameters
NX number of space integration intervals.
DX size of a space integration interval.
NT number of time-integration intervals.
DT size of time-integration intervals.
F initial condition U(0,X) = F(X).
C positive constant.
```

```
Returns
T grid of NT intervals.
X grid of NX intervals.
U solution array.
   [T, X, U] = PDE_HEAT_IMP(N, DX, K, DT, C, F, ALPHA, BETA) solves the heat
equation D U/DT = C D2U/DX2 with the implicit method on [0,1]x[0,1].  Initial
condition is U(0,X) = F and boundary conditions are U(t,0) = ALPHA and U(t,1)
= beta.  C is a positive constant.  N is the number of space integration
intervals and K is the number of time-integration intervals.  DX is the size
of a space integration interval and DT is the size of time-integration intervals.

Parameters
NX number of space integration intervals.
DX size of a space integration interval.
NT number of time-integration intervals.
DT size of time-integration intervals.
F initial condition U(0,X) = F(X).
C positive constant.

Returns
T grid of NT intervals.
X grid of NX intervals.
U solution array.
```

◈The jacobian matrix of the semidiscrete system has eigenvalues between $-4c/(\Delta x)^2$ and 0, and hence the stability region for Euler's method requires that the time step satisfy

$$\Delta t \leq \frac{(\Delta x)^2}{2c} \tag{22}$$

3.10.4 2-D solver for Wave equation

Wave equation :

$$u_{tt} = cu_{xx} \tag{23}$$

```
   [T, X, U] = PDE_WAVE_EXP(N, DX, K, DT, C, F, G, ALPHA, BETA) solves the
wave equation D2U/DT2 = C D2U/DX2 with the explicit method on [0,1]x[0,1].
Initial condition is U(0,X) = F, D U/DT (0,X)= G(X) and boundary conditions
are U(t,0) = ALPHA and U(t,1) = beta.  C is a positive constant.  N is the
number of space integration intervals and K is the number of time-integration
intervals.  DX is the size of a space integration interval and DT is the
size of time-integration intervals.

Parameters
NX number of space integration intervals.
DX size of a space integration interval.
NT number of time-integration intervals.
```

```
DT size of time-integration intervals.
F initial condition U(0,X) = F(X).
G initial condition D U/DT (0,X)= G(X).
C positive constant.

Returns
T grid of NT intervals.
X grid of NX intervals.
U solution array.
   [T, X, U] = PDE_WAVE_IMP(N, DX, K, DT, C, F, G, ALPHA, BETA) solves the
wave equation D2U/DT2 = C D2U/DX2 with the implicit method on [0,1]x[0,1].
Initial condition is U(0,X) = F, D U/DT (0,X)= G(X) and boundary conditions
are U(t,0) = ALPHA and U(t,1) = beta.  C is a positive constant.  N is the
number of space integration intervals and K is the number of time-integration
intervals.  DX is the size of a space integration interval and DT is the
size of time-integration intervals.

Parameters
NX number of space integration intervals.
DX size of a space integration interval.
NT number of time-integration intervals.
DT size of time-integration intervals.
F initial condition U(0,X) = F(X).
G initial condition D U/DT (0,X)= G(X).
C positive constant.

Returns
T grid of NT intervals.
X grid of NX intervals.
U solution array.
```

3.10.5 2-D solver for the Poisson Equation

Poisson equation :

$$u_{xx} + u_{yy} = f(x, y) \tag{24}$$

```
POISSONFD two-dimensional Poisson solver [U, X, Y] = POISSONFD(A, C, B, D,
NX, NY, FUN, BOUND) solves by five-point finite difference scheme the problem
-LAPL(U) = FUN in the rectangle (A,B)x(C,D) with Dirichlet boundery conditions
U(X,Y)=BOUND(X,Y) for any (X, Y) on the boundery of the rectangle.

[U, X, Y, ERROR] = POISSONFD(A,C,B,D,NX,NY,FUN,BOUND,UEX) computes also the
maximum nodal error ERROR with respect to the exact solution UEX. FUN, BOUND
and UEX can be online functions.

Parameters
```

```
A, B
C, D rectangle (A,B)x(C,D) where the solution is computed.
NX X equispaced grid of NX intervals.
NY Y equispaced grid of NY intervals.
FUN
BOUND boundary condition.
UEX exact solution.

Returns
U solution array.
X equispaced grid of NX intervals.
Y equispaced grid of NY intervals.
ERROR maximum nodal error ERROR with respect to the exact solution UEX.
```

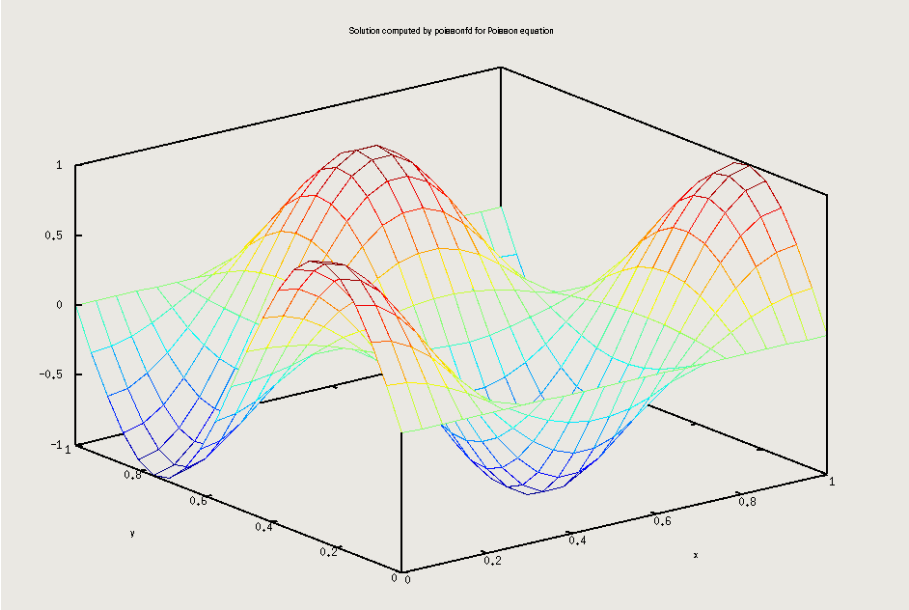

Figure 4: Solution of the Poisson equation with $x \in [0, 1]$ and $t \in [0, 1]$.