Partial Differential Equations

Peter Hertel

Physics Department
Osnabrück University
Germany

Lectures delivered at
TEDA Applied Physics School
Nankai University, Tianjin, PRC

A partial differential equation describes a field, a function of more than one variable. The equation itself is a relation between the arguments, the field and its partial derivatives. A partial differential equation, or PDE, must be augmented by additional conditions which have to be fulfilled by the solution. We here concentrate on numerical aspects. We describe the Finite Difference Method, the Finite Element Method and propagation algorithms. Example PDE solvers are formulated in Matlab, they can easily be generalized to other programming languages and to more complex problems. Since this is a tutorial we restrict ourselves to only two variables.

October 5, 2010

Introduction **3**

1 Finite Differences **4**
1.1 Approximating derivatives 4
1.2 One-dimensional example 6
1.3 Laplace equation on a finite domain 7
1.4 Von Neumann boundary conditions 12
1.5 Infinite domain 16
1.6 Helmholtz equation 22

2 Finite Elements **27**
2.1 Weak form of a partial differential equation 27
2.2 Triangulation and tent functions 28
2.3 A worked out example 31

3 Propagation **33**
3.1 Stability considerations 34
3.2 Crank-Nicholson scheme 36

A Matlab **43**

B Program listing **50**

Introduction

Most laws of physics are relations between derivatives of functions, so called differential equations.

Right from the beginning: Newton's first law states that the product of the mass of a body and its acceleration (second derivative of the location with respect to time) are equal to the force exerted on that body. This law, combined with special initial conditions, allows to calculate the trajectory of a body. This distinction is important. $m\ddot{\boldsymbol{x}} = \boldsymbol{f}$ describe the law. There are many trajectories compatible with that law. The initial conditions $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and $\dot{\boldsymbol{x}}(0) = \boldsymbol{v}_0$ single out a unique solution.

The trajectories of one or many bodies are governed by ordinary differential equations: there is just one independent variable, time in this example.

The temperature field of a body depends on time and space, $T = T(t, \boldsymbol{x})$. Consequently, the time derivative and spatial derivatives (gradients, divergencies, and curls) enter the game, and we have to struggle with partial differential equations.

This series of lectures covers computational methods[1] for solving partial differential equations. The subject is vast, and we must restrict ourselves to simple examples.

We cannot do justice to all methods. Within a time budget of 15 lectures of 90 minutes we concentrate on the finite difference method, introduce the finite element method, and present stability considerations as well as a worked out example for an initial value problem, the propagation of a beam of light in a dielectric waveguide.

The finite element method has won the competition in practically all respects. Although it is difficult to realize from scratch, there are commercial packages which allow for swift solutions.

Examples are presented in the MATLAB programming language. First, because the code is so short that it can be printed and explained within the text. Second, because the code is nevertheless efficient. Analogous programs in Fortran or C++ proved to be neither superior nor better readable, but certainly more difficult to develop.

We apologize for a certain verboseness with respect to program listings. The program code is extracted from the LaTeX sources so that discrepancies between code and documentation will be avoided. The prize to pay is that the entire program has to be printed, in some places with unnecessary details.

[1] as contrasted with analytical methods

1 Finite Differences

The derivative is the limit of a difference quotient,

$$f'(x) = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} \text{ where } \Delta f = f(x + \Delta x/2) - f(x - \Delta/2) \ . \tag{1}$$

All finite difference methods have one feature in common: the derivative $f' = df/dx$ is approximated by the quotient $\Delta f/\Delta x$ of finite differences.
Since a computer has a finite number of different states only, and since the computation time must be finite, it is impossible to work with a continuum of numbers. We have to represent a function by its values at a finite set of representative locations.

1.1 Approximating derivatives

Assume that a function f is represented by values f_j at representative points x_j for which we assume $x_j < x_{j+1}$. What about f' at x_j?
Well, forming the derivative is a linear operation, and the same shall hold for the approximation scheme. We also want to mimic differentiation, so the scheme must be local. We also assume that *a priori* both neighbors are of equal importance. Both requirements are fulfilled if

$$f'(x_j) \approx f'_j = \alpha_- f_{j-1} + \alpha_0 f_j + \alpha_+ f_{j+1} \tag{2}$$

such the the coefficients $\alpha_-, \alpha_0, \alpha_+$ do not depend on the f_j.
Let us interpolate by a quadratic function,

$$\begin{aligned} f(x) \approx p(x) &= \frac{(x - x_j)(x - x_{j+1})}{(x_{j-1} - x_j)(x_{j-1} - x_{j+1})} f_{j-1} \\ &+ \frac{(x - x_{j-1})(x - x_{j+1})}{(x_j - x_{j-1})(x_j - x_{j+1})} f_j \\ &+ \frac{(x - x_{j-1})(x - x_j)}{(x_{j+1} - x_{j-1})(x_{j+1} - x_j)} f_{j+1} \end{aligned} \tag{3}$$

By identifying $f'_j = p'(x_j)$ we arrive at

$$\begin{aligned} \alpha_- &= \frac{x_j - x_{j+1}}{(x_{j-1} - x_j)(x_{j-1} - x_{j+1})} \\ \alpha_0 &= \frac{1}{x_j - x_{j-1}} + \frac{1}{x_j - x_{j+1}} \\ \alpha_+ &= \frac{x_j - x_{j-1}}{(x_{j+1} - x_j)(x_{j+1} - x_{j-1})} \end{aligned} \tag{4}$$

For equidistant spacing, $x_j = jh$, this reduces to

$$f_j' = \frac{f_{j+1} - f_{j-1}}{2h} \quad , \tag{5}$$

a highly plausible result.
From (3) we can also read off an approximation $f_j'' = p''(x_j)$ for f'' at x_j. For the same reasons as explained above we have

$$f''(x_j) \approx f_j'' = \beta_- f_{j-1} + \beta_0 f_j + \beta_+ f_{j+1} \quad . \tag{6}$$

These weights are given by

$$\begin{aligned} \beta_- &= \frac{2}{(x_{j-1} - x_j)(x_{j-1} - x_{j+1})} \\ \beta_0 &= \frac{2}{(x_j - x_{j-1})(x_j - x_{j+1})} \\ \beta_+ &= \frac{2}{(x_{j+1} - x_{j-1})(x_{j+1} - x_j)} \end{aligned} \tag{7}$$

For equidistant spacing we obtain

$$f_j'' = \frac{f_{j-1} - 2f_j + f_{j+1}}{h^2} \quad . \tag{8}$$

This result is highly plausible as well. We should approximate the derivative at $x_j + h/2$ by $(f_{j+1} - f_j)/h$ and at $x_j - h/2$ by $(f_j - f_{j-1})/h$. Their difference, if divided by h, coincides with (8).
We test our findings by the following program. The function to be represented is $f(x) = 1/(1+x)$, its second derivative $f''(x) = 2/(1+x)^3$. We choose an interval $s \in [0, 0.9]$ and/home/phertel map it by $x(s) = s/(1-s)$. This maps $s \in [0, 1)$ to $x \in [0, \infty)$]. Equally spaced s-values lead to a non-equidistant set of representative points x_j.

```
1   clear all;
2   N=32;
3   s=linspace(0,0.9,N);
4   x=s./(1-s);
5   f=1./(1+x);
6   xm=x(1:N-2);
7   x0=x(2:N-1);
8   xp=x(3:N);
9   fm=f(1:N-2);
10  f0=f(2:N-1);
```

```
11  fp=f(3:N);
12  wm=2./(xm-x0)./(xm-xp);
13  w0=2./(x0-xm)./(x0-xp);
14  wp=2./(xp-xm)./(xp-x0);
15  fpp=wm.*fm+w0.*f0+wp.*fp;
16  fdd=2./(1+x0).^3;
17  plot(x0,fdd-fpp,'.','MarkerSize',20);
```

We plot the difference between true and approximated second derivatives in Figure 1.

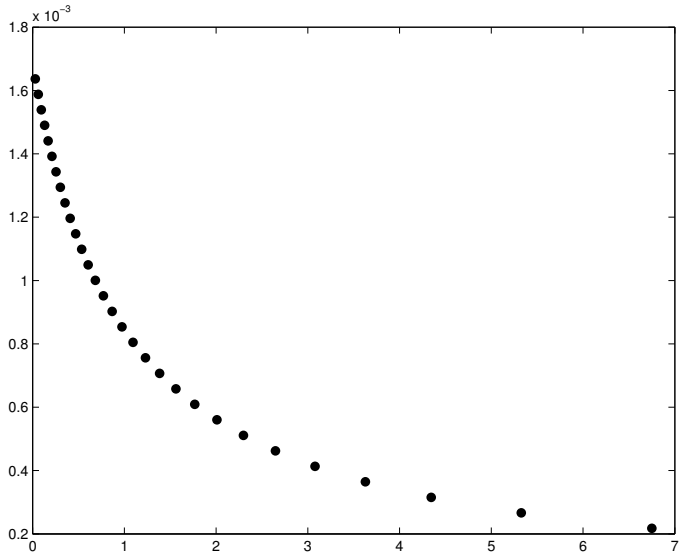

Figure 1: The difference between $f''(x_j)$ and f_j'' is plotted vs. x. The function $f(x) = 1/(1+x)$ was represented at 32 non-equidistant points. Note the 10^{-3} scale.

1.2 One-dimensional example

Let us study a boundary problem on a one-dimensional region, $\Omega = (0, \pi)$ say:

$$u'' + u = 0 \text{ where } u(0) = 0 \ , \ u(\pi) = 1 \tag{9}$$

the solution of which is $u(x) = \sin(x)$.
We divide the interval $[0, \pi]$ into n equal subintervals of length $h = \pi/N$. $x_k = kh$ for $k = 0, 1, \ldots, N$ are representative points. $\partial\Omega = \{x_0, x_n\}$ make up the boundary while $\{x_1, x_2, \ldots, x_{n-1}\}$ represents the interior of Ω. We

approximate the derivative at $x_k - h/2$ by $(x_k - x_{k-1})/h$ and at $x_k + h/2$ by $(x_{k+1} - x_k)/h$. The symmetric second derivative at x_k should therefore be approximated by

$$u_k'' = \frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} \quad . \tag{10}$$

These are the equations to be solved for $k = 1, 2, \ldots, n-1$:

$$u_{k+1} - 2u_k + u_{k-1} + h^2 * u_k = 0 \text{ where } u_0 = 0 \text{ and } u_n = 1. \quad . \tag{11}$$

Note that this is a system of linear equations $Mu = r$. The matrix M has $-2 + h^2$ on its diagonal and 1 in both side diagonals. The right hand side r vanishes except for $r_{n-1} = -u_n = -1$.

Here is the realization in MATLAB:

```
1   N=16;
2   h=0.5*pi/(N-1);
3   d0=ones(N-2,1);
4   d1=ones(N-3,1);
5   M=diag(d1,-1)+(h*h-2)*diag(d0,0)+diag(d1,1);
6   r=zeros(N-2,1);
7   r(N-2)=-1;
8   u=M\r;
9   x=linspace(0,pi/2,N);
10  uu=[0;u;1];
```

`diag(v,k)` creates a matrix with a diagonal `v` which is `k` places off the main diagonal. `u=M\r` solves the system of linear equations $Mu = r$. Figure 2 depicts the result.

For N as small as 16 (i. e. 14 variables) the maximum deviation is less than 3×10^{-4}, for $N = 1024$ the approximation is better than 10^{-7}. M then requires more than 8 MB of memory, and the problem is solved in 0.22 s on my new laptop (as of 2008).

1.3 Laplace equation on a finite domain

Let us study a very simple example. An inner tube of square cross section is placed concentrically within an outer tube of likewise square cross section. The inner tube is kept at constant temperature T_i, the outer tube a temperature T_o. Between the two tubes there is a homogeneous heat conducting medium. We are interested in the temperature distribution T between the two tubes in a stationary state. The temperature field T then does not depend on time and has to fulfill $(\partial_x^2 + \partial_y^2)\, T = 0$, the Laplace[2] equation.

[2]Pierre-Simon Laplace (1749-1827), French mathematician, astronomer, and physicist

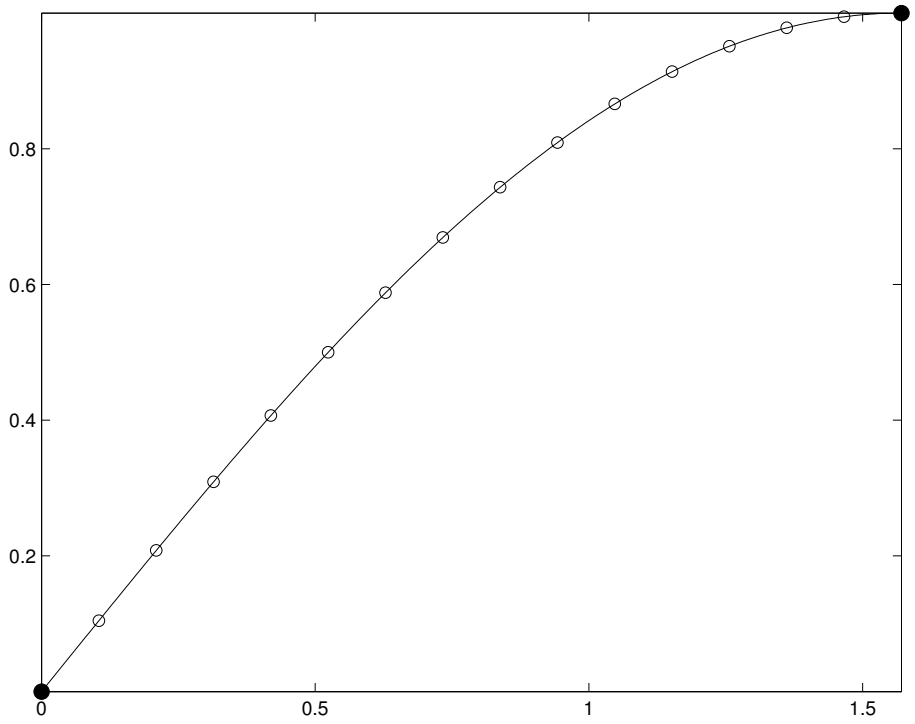

Figure 2: Open circles represent the solution of $u'' + u = 0$ on $(0, \pi/2)$ by the finite difference method with suitable boundary conditions. The values at 14 representative points are contrasted with the exact solution. Full circles represent the boundary conditions.

Without loss of generality we define the heat conducting region by

$$\Omega = (-1,1) \times (-1,1) - [-a,a] \times [-a,a] \ . \tag{12}$$

The tubes have a width and height of 2 and $2a$, respectively ($0 < a < 1$). The boundary $\partial\Omega$ consists of two disjoint sets. The inner part is

$$\partial\Omega_{\rm i} = \{(x,y) \mid (|x| = a \ \text{ and } \ |y| \le a) \ \text{ or } \ (|y| = a \ \text{ and } \ |x| \le a) \} \ . \tag{13}$$

The same expression with a replaced by 1 defines the outer part $\partial\Omega_{\rm o}$ of the boundary.
We define u by $T = T_{\rm o} + u(T_{\rm i} - T_o)$. The function $u = u(x,y)$ has to fulfill[3]

$$u_{xx} + u_{yy} = 0 \ \text{ on } \ \Omega \ , \tag{14}$$

[3] u_x is short for $\partial_x u$ which is short for $\partial u(x,y)/\partial_x$. u_{xx} etc. have to been read likewise.

$$u = 0 \text{ on } \partial\Omega_{\rm o} \ , \tag{15}$$

$$u = 1 \text{ on } \partial\Omega_{\rm i} \ . \tag{16}$$

We here assume equidistant spacing, the same for both coordinates. The Laplacian at $(x_j, y_k) = (jh, kh)$ is then approximated by

$$(\Delta u)_{j,k} = \frac{-4u_{j,k} + u_{j+1,k} + u_{j,k+1} + u_{j-1,k} + u_{j,k-1}}{h^2} \ . \tag{17}$$

The $u_{j,k}$ are either variables (if $a = (j,k)$ indexes an interior point), or boundary values. $a = 1, 2, \ldots, N_{\rm v}$ is a running index for the N_v variables.

We first setup a matrix `w` which defines the computational window. Its entries are the prescribed value of u, or `Inf` which signals 'not yet computed', or 'a variable'.

```
1   N=32;
2   x=linspace(-1,1,N);
3   y=linspace(-1,1,N);
4   a=0.25;
5   [X,Y]=meshgrid(x,y);
6   w([1:N],[1:N])=Inf;
7   w(abs(X)==1|abs(Y)==1)=0;
8   w(abs(X)<=a&abs(Y)<=a)=1;
9   u=laplace(w);
10  mesh(u);
```

The job is done by the following function. `u=laplace(w)` solves the Laplace equation on a domain described by `w`. We have assumed equidistant spacing, so that h in (17) drops out.

```
1   function u=laplace(w)
2   [Nx,Ny]=size(w);
3   jj=zeros(Nx*Ny,1);
4   kk=zeros(Nx*Ny,1);
5   aa=zeros(Nx,Ny);
6   Nv=0;
7   for j=1:Nx
8      for k=1:Ny
9         if w(j,k)==Inf
10           Nv=Nv+1;
11           jj(Nv)=j;
12           kk(Nv)=k;
13           aa(j,k)=Nv;
```

```
14          end;
15       end;
16    end;
```

This part of the program finds out the number of variables and establishes the mapping $(j,k) \to a$ as well as $a \to (j,k)$. Next we declare a so far empty sparse square matrix of dimension N_{v} as well as a right hand side column vector.

We run through the rows of the Laplacian matrix, `for a=1:Nv`. The diagonal, `lap(a,a)` is set to -4. Then we inspect the four neighbors `b` to the east, north, west, and south. If the neighbor is a variable, `lap(a,b)` is set to 1. If it is a boundary value, it is added to the right hand side `rhs(a)`

```
17    lap=sparse(Nv,Nv);
18    rhs=zeros(Nv,1);
19    for a=1:Nv
20       lap(a,a)=-4;
21       j=jj(a);
22       k=kk(a);
23       if w(j+1,k)==Inf
24          b=aa(j+1,k);
25          lap(a,b)=1;
26       else
27          rhs(a)=rhs(a)-w(j+1,k);
28       end;
29       if w(j,k+1)==Inf
30          b=aa(j,k+1);
31          lap(a,b)=1;
32       else
33          rhs(a)=rhs(a)-w(j,k+1);
34       end;
35       if w(j-1,k)==Inf
36          b=aa(j-1,k);
37          lap(a,b)=1;
38       else
39          rhs(a)=rhs(a)-w(j-1,k);
40       end;
41       if w(j,k-1)==Inf
42          b=aa(j,k-1);
43          lap(a,b)=1;
44       else
45          rhs(a)=rhs(a)-w(j,k-1);
46       end;
47    end;
```

It remains to solve `lap*v=rhs` for `v` which is the main effort[4]. The resulting vector of values for the variables must then be inserted into the matrix `u(j,k)` representing the field.

```
48   v=lap\rhs;
49   u=w;
50   for a=1:Nv
51      u(jj(a),kk(a))=v(a);
52   end;
```

We have visualized the result in Figure 3.

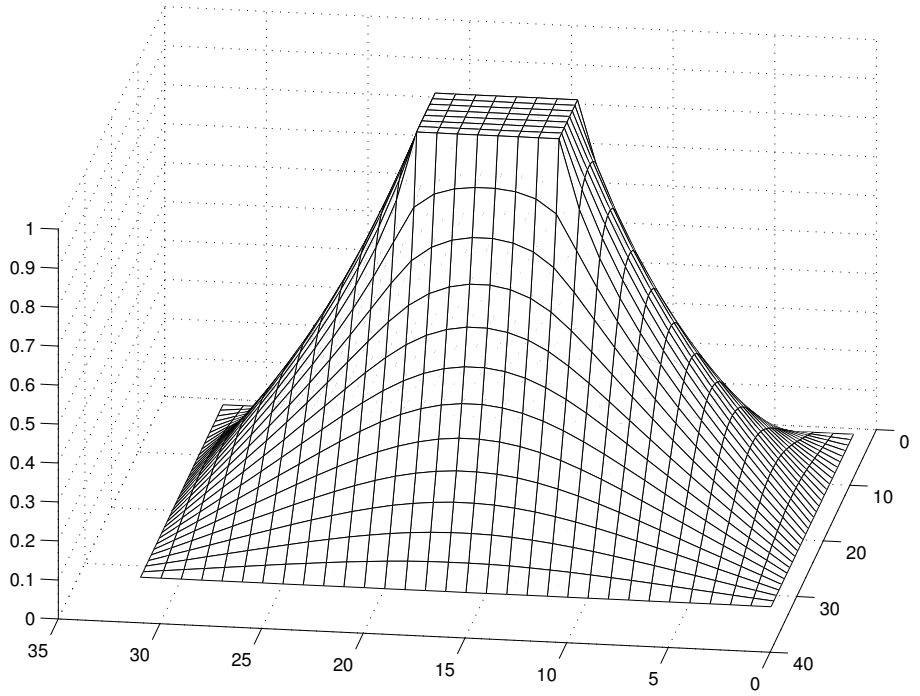

Figure 3: The interior rectangular tube is kept at (a relative) temperature $u = 1$, the outer at $u = 0$. We have plotted the temperature in-between. The x, y axes are labeled by the indices of the computational window.

At this point we should discuss performance issues.
The 32×32 computational window amounts to 836 variables, the sparse Laplacian requires 52 kB of storage. The computation time is 0.2 s on a standard PC (as of 2008). A 128×128 window results in 14,852 variables

[4]measured in lines of source code, and, for large problems, also in computation time

with a Laplacian of 0.94 MB, if stored as a sparse matrix. A full matrix would need 1.8 GB. The computation time is now 4 s on a standard PC.

1.4 Von Neumann boundary conditions

The example of subsection 1.3 has an obvious symmetry: $u(x,y) = u(-x,y)$ and $u(x,y) = u(x,-y)$. Therefore, it suffices to study the problem on the upper right quadrant. Because u is an even function of x and y, the respective partial derivatives have to vanish at $x = 0$ or $y = 0$.

Dirichlet[5] boundary conditions specify the function value at the boundary or parts of the boundary. Our previous example was formulated with Dirichlet boundary conditions.

One may also prescribe the normal derivative at the boundary or parts of the boundary. One speaks of von Neumann[6] boundary conditions in this case.

In our case we have to solve the Laplace equation $u_{xx} + u_{yy} = 0$ on

$$\Omega = [0,1) \times [0,1) - [0,a] \times [0,a] \ . \tag{18}$$

The boundary conditions to be observed are

$$\begin{aligned}
a < x < 1 \ , \ y = 0 \quad &: \quad u_y = 0 \\
x = 0 \ , \ a < y < 1 \quad &: \quad u_x = 0 \\
x = a \ , \ 0 \le y \le a \quad &: \quad u = 1 \\
0 \le x \le a \ , \ y = a \quad &: \quad u = 1 \\
0 \le x \le 1 \ , \ y = 1 \quad &: \quad u = 0 \\
x = 1 \ , \ 0 \le y \le 1 \quad &: \quad u = 0
\end{aligned} \tag{19}$$

We now have to distinguish between three sorts of quantities: fixed values, variables representing the field at interior points, and variables at a boundary being restricted by a vanishing gradient.

Again we describe the problem by a matrix `w` representing the computational window[7]. Each entry is either a number (representing a value of the field at a boundary) or a symbol. The symbols are `IP` (interior point), `EA` (the

[5] Peter Gustav Lejeune Dirichlet (1805-1859), German mathematician, professor at Breslau, Berlin and Göttingen

[6] Johann von Neumann (1903-1957), German-American mathematician, lecturer at Berlin and Hamburg, professor at Princeton university, later at the Institute of Advanced Studies at Princeton.

[7] We treat `j` in `w(j,k)` as the x variable running from left to right, or from west to east. `k` corresponds to the variable y running from bottom to top, or from south to north.

gradient towards east has to vanish) and likewise `NO`, `WE`, and `SO`. There is one more symbol `VA` such that `w(j,k)>VA` is true if the field value at (j,k) is a variable. Needless to say that the boundary values must be less than the symbols.

If (j,k) is characterized by `EA`, we know that $(j+1,k)$ is outside the domain of definition. However,

$$(u_x)_{j,k} = \frac{u_{j+1,k} - u_{j-1,k}}{2h} \tag{20}$$

must vanish, so that the Laplacian of u at (j,k) has to be calculated with $u_{j+1,k} = u_{j-1,k}$. With $a = (j,k)$ and $b = (j-1,k)$ we have to enter $(\Delta)_{a,b} = 2/h^2$ instead of $1/h^2$ if (j,k) and $(j-1,k)$ were both internal points.

The following code sets up the computational window `w`, calls `u=lap_vn(w)` and generates a mesh representation of the result.

```
1   global VA IP EA NO WE SO
2   VA=1000;
3   IP=1001;
4   EA=1002;
5   NO=1003;
6   WE=1004;
7   SO=1005;
8   M=9;
9   N=33;
10  w([1:N],[1:N])=IP;
11  w(1,[M+1:N-1])=WE;
12  w([M+1:N-1],1)=SO;
13  w([1:M],[1:M])=1;
14  w(N,:)=0;
15  w(:,N)=0;
16  u=lap_vn(w);
17  mesh(u);
```

Here is how the Laplace equation with both Dirichlet and von Neumann boundary conditions is solved.

For technical reasons (because we want to use it as a global variable) the computational window has to be duplicated. As before, we establish the mapping $(j,k) \leftrightarrow a$ and reserve space for the sparse matrix `lap` and the right hand side `rhs`.

```
1   function u=lap_vn(w);
2   global VA IP EA NO WE SO
```

```
3   global ww lap rhs a aa
4   ww=w;
5   [Nx,Ny]=size(ww);
6   jj=zeros(Nx*Ny,1);
7   kk=zeros(Nx*Ny,1);
8   aa=zeros(Nx,Ny);
9   Nv=0;
10  for j=1:Nx
11     for k=1:Ny
12        if ww(j,k)>VA
13           Nv=Nv+1;
14           jj(Nv)=j;
15           kk(Nv)=k;
16           aa(j,k)=Nv;
17        end
18     end
19  end
20  lap=sparse(Nv,Nv);
21  rhs=zeros(Nv,1);
```

Next we loop over the rows of the Laplacian.

```
22  for a=1:Nv
23     lap(a,a)=-4;
24     j=jj(a);
25     k=kk(a);
```

When inspecting neighbors, we have to distinguish between different cases: does the variable represent an interior point, or is it restricted by a von Neumann condition?

```
26     switch ww(j,k)
27        case(IP)
28           neighbor(j+1,k);
29           neighbor(j,k+1);
30           neighbor(j-1,k);
31           neighbor(j,k-1);
```

The function `neighbor` is explained below. We now handle the case that the normal derivative towards the east has to vanish:

```
32        case(EA)
33           neighbor(j,k+1);
34           neighbor(j,k-1);
35           lap(a,aa(j-1,k))=2;
```

and then the same with north, west, and south:

```
36          case(NO)
37             neighbor(j+1,k);
38             neighbor(j-1,k);
39             lap(a,aa(j,k-1))=2;
40          case(WE)
41             neighbor(j,k+1);
42             neighbor(j,k-1);
43             lap(a,aa(j+1,k))=2;
44          case(SO)
45             neighbor(j+1,k);
46             neighbor(j-1,k);
47             lap(a,aa(j,k+1))=2;
48       end % switch
49    end % for a...
```

After these preliminaries, the system of linear equations has to be solved, and the solution must be placed into the u-field. minput('edit graphics and continue by striking a key');

```
50    v=lap\rhs;
51    u=ww;
52    for a=1:Nv
53       u(jj(a),kk(a))=v(a);
54    end
```

It remains to formulate the `neighbor` function:

```
55    function neighbor(j,k)
56    global ww lap rhs a aa VA
57    if ww(j,k)>VA
58       lap(a,aa(j,k))=1;
59    else
60       rhs(a)=rhs(a)-ww(j,k);
61    end
```

The result is visualized in Figure 4.

In general, a von Neumann boundary condition prescribes the normal derivative u_{n} which must not necessarily vanish. For an `EA` boundary condition this would mean

$$u_{j+1,k} = u_{j-1,k} + 2h(u_{\mathrm{n}})_{j,k} \tag{21}$$

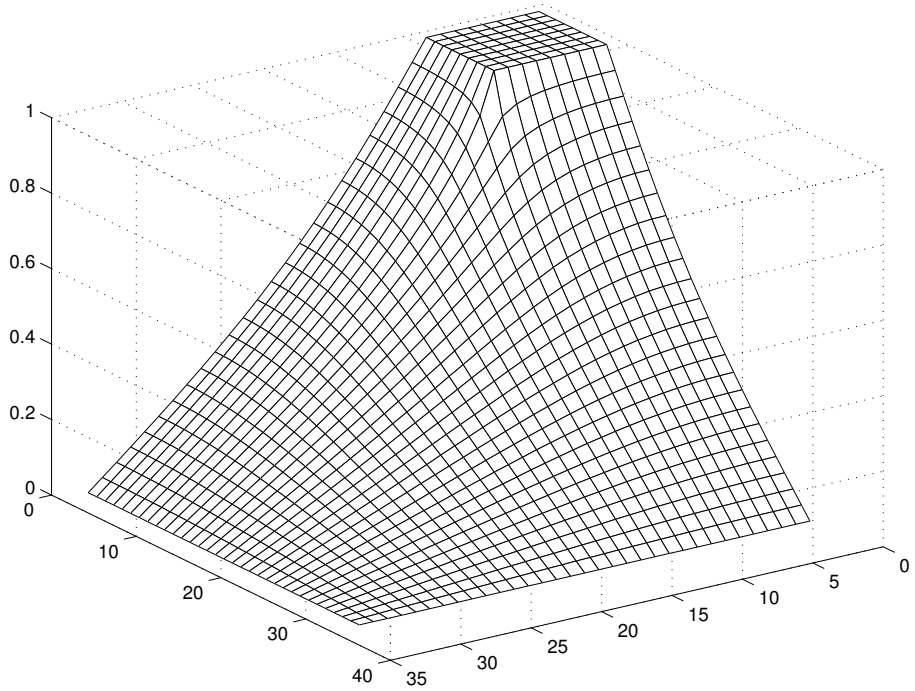

Figure 4: See Figure 3 for details. Exploiting symmetry, only one quadrant of the previous computational window was investigated. Dirichlet as well as von Neumann boundary conditions had to be observed.

to be inserted into the approximate Laplacian

$$(\Delta u)_{j,k} = \frac{-4u_{j,k} + u_{j+1,k} + u_{j,k+1} + u_{j-1,k} + u_{j,k-1}}{h^2} \quad . \tag{22}$$

It is left as a challenge to the reader to exploit the $x \leftrightarrow y$ symmetry as well.

1.5 Infinite domain

Let us discuss another problem. Two parallel infinitely long conducting strips are held at constant potential $u = +1$ and $u = -1$, respectively. There is another boundary condition: the potential ϕ has to vanish at infinity. Now, infinity is too large, so we represent it by a finite value, see Figure 5. For symmetry reasons it suffices to calculate the potential in the upper right quadrant. After all, the computational window must be large, so we should use variables sparingly.

The following program solves the Laplace equation $u_{xx} + u_{yy} = 0$ on the first quadrant. `SX` and `SY` define half a strip, `M` is a margin which should be

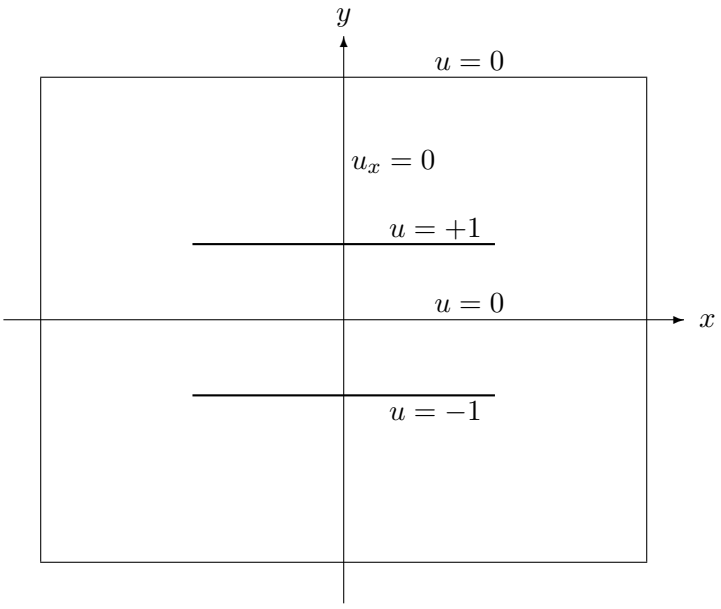

Figure 5: The computational window for a strip capacitor.

infinite.

```
1   global VA IP EA NO WE SO
2   VA=1000;
3   IP=1001;
4   EA=1002;
5   NO=1003;
6   WE=1004;
7   SO=1005;
8   SX=14;
9   SY=7;
10  M=20;
11  NX=SX+M;
12  NY=SY+M;
13  w([2:NX-1],[2:NY-1])=IP;
14  w(1,[1:SY-1,SY+1:NY-1])=WE;
15  w(1:SX,SY)=1;
16  w(:,1)=0;
17  w(NX,:)=0;
18  w(:,NY)=0;
19  u=lap_vn(w);
20  contour(u',linspace(0,1,24));
21  axis equal;
```

```
22   axis([0,4,0,4]);
23   axis tight;
```

The result is plotted as Figure 6.

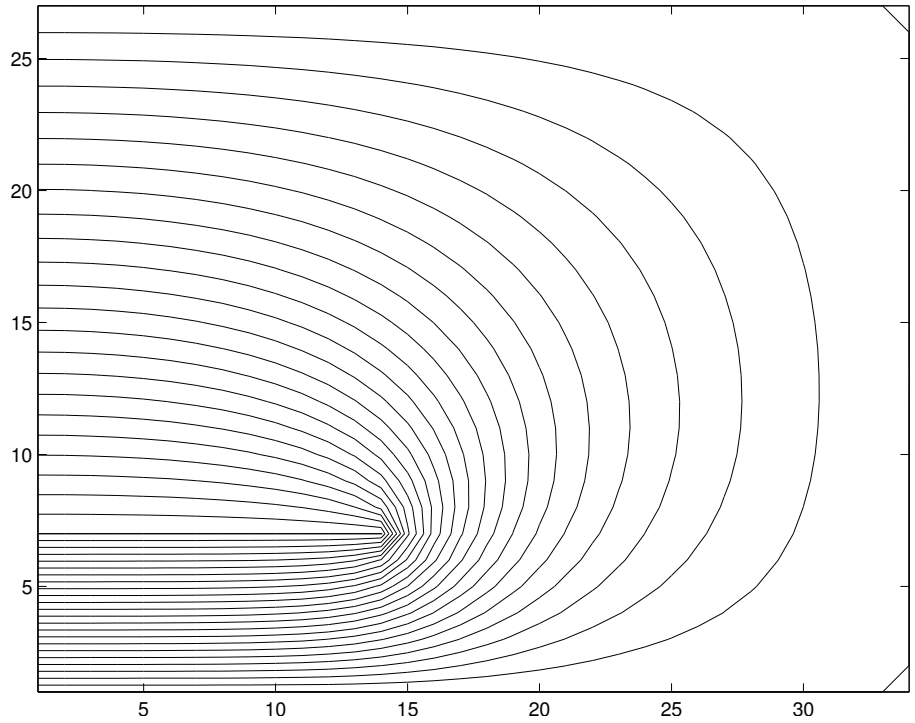

Figure 6: The potential of a strip capacitor in the first quadrant. There are 811 variables. The axes are labeled by the variable index.

Between the plates well away from the edges the electric field strength $\boldsymbol{E} = (-u_x, -u_y)$ is constant. However, the simulation of free space is rather bad. The field is obviously pressed upon by the too close-by boundary representing infinity.

Increasing the margin `M` helps, but not much because the computational time grows excessively with the number of variables.

A way out is to drop the equidistant spacing scheme. The region close to infinity can be represented by a coarser resolution than the neighborhood of the strips. Since we do not want to introduce more than one complication at the same time, both coordinate axes are treated alike.

We map by

$$z = \frac{s}{1-s} \quad \text{or} \quad z = s/(1-s) \tag{23}$$

between $[0, 1)$ and $[0, \infty)$.
Recall (7) how to work out the Laplacian with non-equidistant spacing.
The following code is very special to our example of a strip capacitor, the result is displayed in Figure 7.

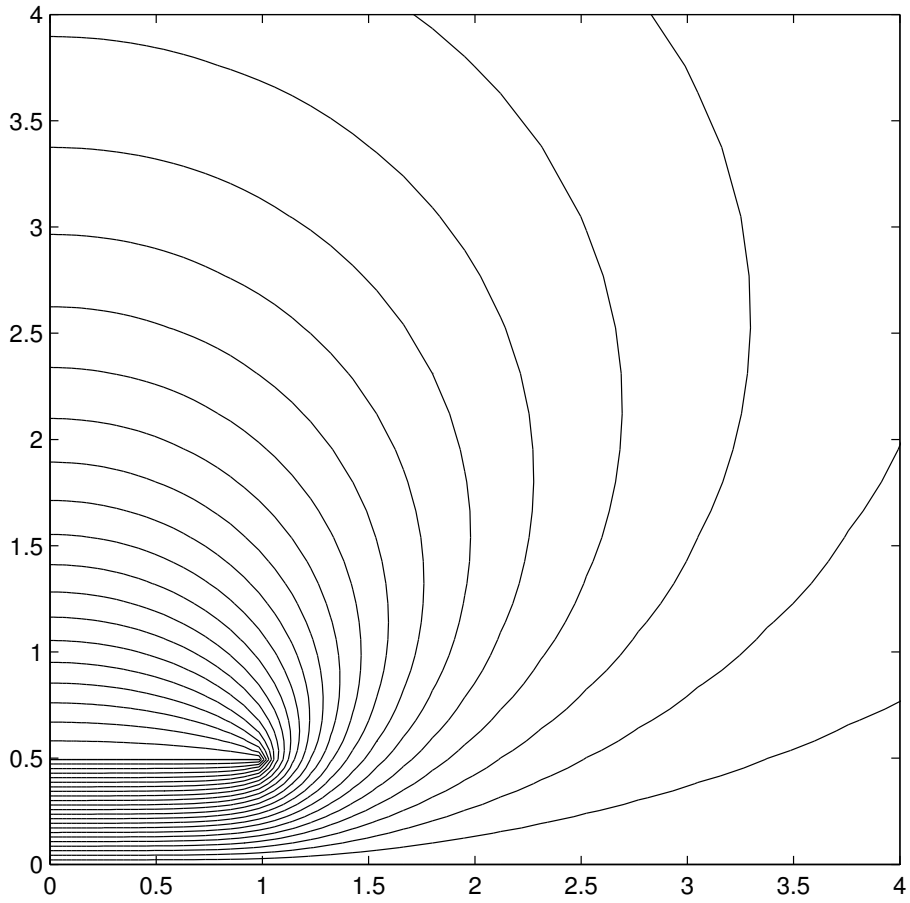

Figure 7: Lines of constant potential for a strip capacitor. By properly mapping the coordinates, the boundary condition $u = 0$ at infinity was simulated much better than in Figure 6.

```
1  xs=1;
2  ys=0.5;
3  N=50;
4  s=linspace(0,0.9,N);
5  z=s./(1-s);
6  [~,XS]=min(abs(z-xs));
```

```
 7   [~,YS]=min(abs(z-ys));
 8   zz=[-z(2),z];
 9   zm=zz(1:N-1);
10   z0=zz(2:N);
11   zp=zz(3:N+1);
12   betm=2./(zm-z0)./(zm-zp);
13   bet0=2./(z0-zm)./(z0-zp);
14   betp=2./(zp-zm)./(zp-z0);
```

This part defines the strip ($0 \le x \le x_s$ and $y = y_s$), defines the mapping, and calculates the weights $\beta_-, \beta_0, \beta_+$.
The next piece determines indexing $j, k \leftrightarrow a$:

```
15   aa=zeros(N,N);
16   jj=zeros(1,N*N);
17   kk=zeros(1,N*N);
18   V=0;
19   for j=1:N-1
20      for k=2:N-1
21         if ((j==0)&&(k~=YS))|((j<=XS)&&(k~=YS))|(j>XS)
22            V=V+1;
23            aa(j,k)=V;
24            jj(V)=j;
25            kk(V)=k;
26         end
27      end
28   end
```

Now we set up the sparse Laplacian matrix `lap` and the right hand side `rhs`:

```
29   lap=sparse(V,V);
30   rhs=zeros(V,1);
31   for a=1:V
32      j=jj(a);
33      k=kk(a);
34      lap(a,a)=bet0(j)+bet0(k);
35      if j+1<N
36         lap(a,aa(j+1,k)) =betp(j);
37      end
38      if (k+1==YS)&&(j<=XS)
39         rhs(a)=-betp(k);
40      elseif k+1<N
41         lap(a,aa(j,k+1))=betp(k);
42      end
```

```
43      if j-1==0
44         lap(a,aa(j+1,k))=lap(a,aa(j+1,k))+betp(j);
45      elseif (k==YS)&&(j-1==XS)
46         rhs(a)=-betm(j);
47      else
48         lap(a,aa(j-1,k))=betm(j);
49      end;
50      if (k-1==YS)&&(j<=XS)
51         rhs(a)=-betm(k);
52      elseif k-1>1
53         lap(a,aa(j,k-1))=betm(k);
54      end
55   end
```

If $j = 1$ we have to implement the von Neumann boundary condition $u_x = 0$ which is done by identifying the fictitious variable $u_{0,k}$ with $u_{2,k}$.
We now solve the sparse system of linear equations and place the solution at the proper place into the matrix u representing the potential:

```
56   v=lap\rhs;
57   u=zeros(N,N);
58   u([1:XS],YS)=1;
59   for a=1:V
60      u(jj(a),kk(a))=v(a);
61   end
```

The remaining part of the program visualizes the result:

```
62   contour(z,z,u',linspace(0,1,24));
63   axis equal;
64   axis([0,4,0,4]);
65   print -deps2 fdid2.eps
```

There were 2324 variables in this example. On my old 266 MHz laptop the program required six seconds for setting up the Laplacian matrix and the right hand side, and less than one second for solving the system of linear equations. The sparse Laplacian matrix requires only 15 kB of storage. A full matrix would have occupied 43.2 MB. Put otherwise, without sparse matrix algorithms, this or only slightly larger problems would barely be tractable. You should also have learnt that only a non-equidistant spacing scheme allowed to keep the number of variables reasonably small when a boundary at infinity is involved.

1.6 Helmholtz equation

Maxwell's equations read

$$\boldsymbol{\nabla} \cdot \boldsymbol{D} = \rho \ , \ \boldsymbol{\nabla} \cdot \boldsymbol{B} = 0 \ , \ \boldsymbol{\nabla} \times \boldsymbol{E} = -\dot{\boldsymbol{B}} \ , \ \boldsymbol{\nabla} \times \boldsymbol{H} = \boldsymbol{j} + \dot{D} \ . \quad (24)$$

$\boldsymbol{D}$ is the dielectric displacement, $\boldsymbol{B}$ the induction, $\boldsymbol{E}$ the electric and $\boldsymbol{H}$ the magnetic field strength. The electromagnetic field is incited by a charge density ρ and a current density $\boldsymbol{j}$.
A non-magnetic isotropic dielectric medium is characterized by

$$\boldsymbol{D} = \epsilon\epsilon_0 \boldsymbol{E} \ \text{ and } \ \boldsymbol{B} = \mu_0 \boldsymbol{H} \quad (25)$$

where $\mu_0\epsilon_0 c^2 = 1$. c is the vacuum speed of light, ϵ the frequency dependent permittivity (or relative dielectric constant).
We discuss the propagation of light with angular frequency ω. Maxwell's equations reduce to

$$\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \boldsymbol{E} = k_0^2 \epsilon \boldsymbol{E} \ , \quad (26)$$

where $\omega = k_0^2 c$. If

$$0 = \boldsymbol{\nabla} \times \epsilon \boldsymbol{E} \approx \epsilon \boldsymbol{\nabla} \cdot \boldsymbol{E} \quad (27)$$

is an acceptable approximation, (26) reduces to

$$-\Delta \boldsymbol{E} = k_0^2 \epsilon \boldsymbol{E} \ , \quad (28)$$

the Helmholtz[8] equation.
Here we want to study a dielectric waveguide which is described by a permittivity profile $\epsilon = \epsilon(x, y)$. Such a waveguide carries modes of the form

$$\boldsymbol{E}(t, x, y, z) = \boldsymbol{\mathcal{E}}(x, y)\, e^{\,i\beta z - i\omega t} \ . \quad (29)$$

Each component u of $\boldsymbol{\mathcal{E}}$ has to obey

$$u_{xx} + u_{yy} + k_0^2\, \epsilon\, u = \beta^2\, u \ . \quad (30)$$

We look for guided waves which are characterized by

$$\int dxdy\, |u(x, y)|^2 < \infty \ , \quad (31)$$

which means finite energy flux. Square integrable solutions of (30) are possible for certain discrete eigenvalues $\Lambda = \beta^2$ only.

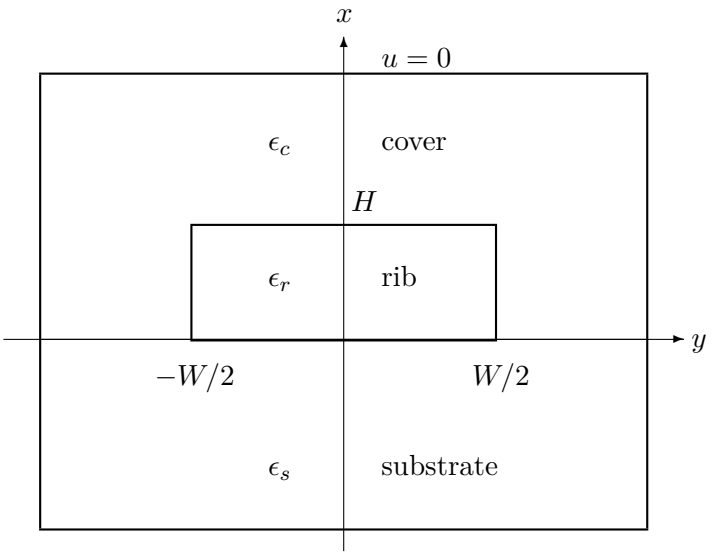

Figure 8: The computational window for a rib waveguide.

We will discuss one of the simplest devices, a rib waveguide. On top of a substrate (permittivity ϵ_s) there is a rib of rectangular cross section (width W, height H, permittivity ϵ_r) surrounded by a cover (permittivity ϵ_c). See Figure 8 for a sketch.

The linear operator defined by the left hand side of (30) is simple to construct. It is the Laplacian plus a contribution on the diagonal. Let's do it!

We place the computational window `[1:NX,1:NY]` such that `x` runs from `xlo` to `xhi` and `y` from `ylo` to `yhi`.

```
1   EC=1.00;
2   ES=3.80;
3   ER=5.20;
4   lambda=1.50;
5   k0=2*pi/lambda;
6   hx=0.05;
7   hy=0.05;
8   xlo=20;
9   xhi=35;
10  ylo=15;
11  yhi=45;
```

[8] Hermann von Helmholtz (1821-1894), German physicist and physiologist.

```
12   NX=50;
13   NY=59;
```

The permittivity profile is crudely defined by

```
14   prm=EC*ones(NX,NY);
15   prm([1:xlo-1],:)=ES*ones(xlo-1,NY);
16   prm([xlo+1:xhi-1],[ylo+1:yhi-1])=ER*ones(xhi-xlo-1,yhi-ylo-1);
```

We want to avoid continuity considerations here and therefore refine the permittivity profile such that it is the average at the border of two different media:

```
17   prm(xlo,:)=0.5*(prm(xlo-1,:)+prm(xlo+1,:));
18   prm(xhi,[ylo+1:yhi-1])=0.5*(ER+EC)*ones(1,yhi-ylo-1);
19   prm([xlo+1:xhi-1],[ylo,yhi])=0.5*(ER+EC)*ones(xhi-xlo-1,2);
```

At corners we set the average of the four neighboring quadrants,

```
20   prm(xlo,[ylo,yhi])=0.25*(2*ES+EC+ER);
21   prm(xhi,[ylo,yhi])=0.25*(3*EC+ER);
```

The following line calls a function which sets up the Helmholtz operator, the left hand side of (30):

```
22   H=helmholtz(hx,hy,k0*k0*prm);
```

The corresponding program will be discussed later.

Now, the Helmholtz operator, which is defined by the left hand side of (30), is the sum of the Laplacian and a multiplication operator. The Laplacian is a negative operator, and the eigenvalues of the Helmholtz operator must be below the maximal permittivity. We look for the highest and the next lower value defining the ground mode and the first excited mode. The string `'LA'` tells the diagonalization algorithm to look for eigenvalues with the largest algebraic part, in this case, two of them.

```
23   [u,d]=eigs(H,2,'LA');
24   mode1=reshape(u(:,1),NX,NY);
25   mode2=reshape(u(:,2),NX,NY);
```

We display the ground mode in Figure 9 and the first excited mode in Figure 10.

After what we have studied before, setting up the Helmholtz matrix is simple:

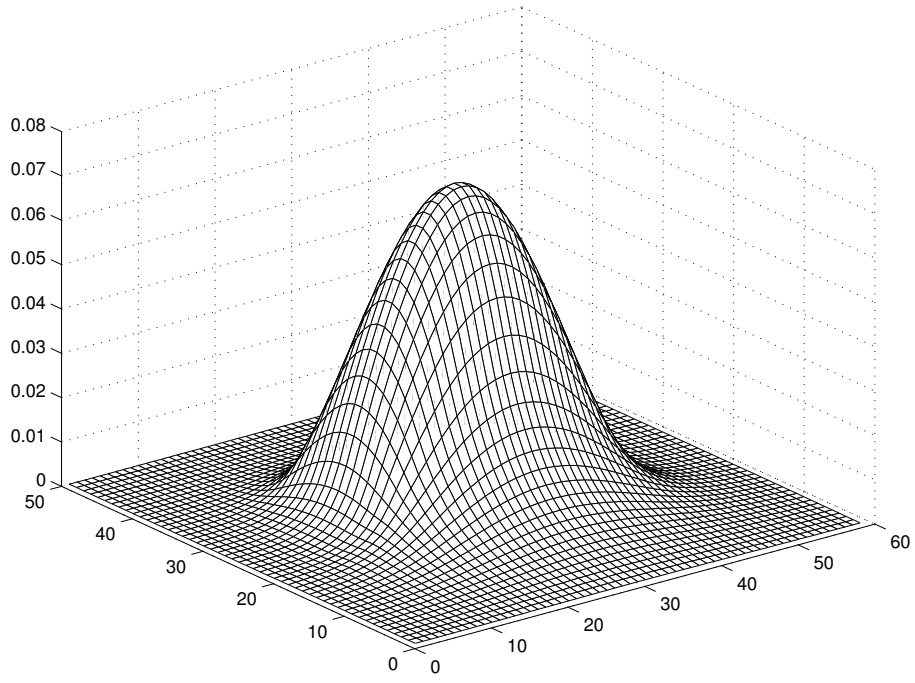

Figure 9: The ground mode of a dielectric waveguide. The electric field strength is plotted on the waveguide cross section. The cover, substrate, and rib permittivities are $\epsilon_c = 1.00$, $\epsilon_s = 3.80$, and $\epsilon_r = 5.20$. The light wavelength is $\lambda = 1.50$ microns. The rib is 0.75 microns high and 1.5 microns wide.

```
1   function H=helmholtz(hx,hy,d)
2   [NX,NY]=size(d);
3   N=NX*NY;
4   ihx2=1/hx/hx;
5   ihy2=1/hy/hy;
6   md=-2*(ihx2+ihy2)*ones(N,1)+reshape(d,NX*NY,1);
7   xd=ihx2*ones(N,1);
8   yd=ihy2*ones(N,1);
9   H=spdiags([yd,xd,md,xd,yd],[-NX,-1,0,1,NX],N,N);
10  for n=NX:NX:N-NX
11     H(n,n+1)=0;
12     H(n+1,n)=0;
13  end
```

The last four lines of code suppress links between variables at the lower and upper border of the computational window.

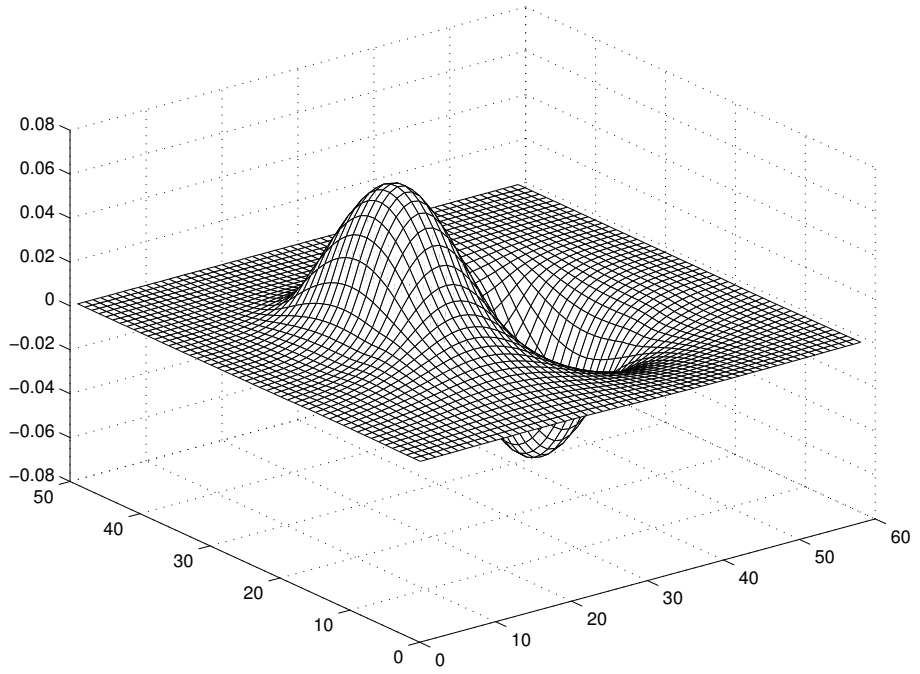

Figure 10: The first excited mode of a dielectric waveguide. See Figure 9 for a description of the waveguide. The structure cannot guide more than two modes.

We will not discuss further details of waveguiding here. In particular, polarization and continuity requirements at the interface of different media pose problems which are beyond the scope of this text.

2 Finite Elements

In the preceding section we have discussed linear partial differential equations $Lu = 0$ or $Lu = \Lambda u$. The function u was approximated by representative values, i. e. by a vector, the differential operator L by a matrix where non-vanishing boundary values lead to a right hand side.

The finite element method expands u into a set of functions such that the operator L can be worked out. The residual $r = Lu$ has to vanish. Instead of solving for $r = 0$ directly we demand that the scalar products (f, r) vanish for a sufficiently large set of test functions f. The finite element method proper makes use of tent functions defined on a triangulation of the domain Ω.

2.1 Weak form of a partial differential equation

We study the class

$$-\boldsymbol{\nabla} c \boldsymbol{\nabla} u + a\, u = f \tag{32}$$

of elliptic partial differential equations. $u = u(x, y)$ is the searched for solution. It is defined in the interior of a finite domain $\Omega \subset \mathbb{R}^2$. $\boldsymbol{\nabla} = (\partial_x, \partial_y)$ is the nabla operator in two dimensions. All previously discussed examples belong to this class. c, a, and f are complex valued functions $\Omega \to \mathbb{C}$, just as u.

At the boundary $\partial\Omega$ we demand

$$\boldsymbol{n} c \boldsymbol{\nabla} u + q\, u = g \ , \tag{33}$$

$\boldsymbol{n}$, the outward pointing normal vector, q, and g are defined on the boundary $\partial\Omega$. If the first term is dropped, we speak of a Dirichlet boundary conditions. $q = 0$ stands for a von Neumann boundary conditions. We allow for both.

Let us choose an arbitrary test function $v : \Omega \to \mathbb{C}$. With $d\Omega = dxdy$ we may write (32) as

$$-\int_\Omega d\Omega\, v \boldsymbol{\nabla} c \boldsymbol{\nabla} u + \int_\Omega d\Omega\, vau = \int_\Omega d\Omega\, vf \tag{34}$$

which, after partial integration, yields

$$-\int_{\partial\Omega} ds\, \boldsymbol{n} vc \boldsymbol{\nabla} u + \int_\Omega d\Omega\, (\boldsymbol{\nabla} v)\, c\, (\boldsymbol{\nabla} u) + \int_\Omega d\Omega\, vau = \int_\Omega d\Omega\, vf \ . \tag{35}$$

We insert the boundary condition (33) and find

$$\int_\Omega d\Omega\, (\boldsymbol{\nabla} v)\, c\, (\boldsymbol{\nabla} u) + \int_\Omega d\Omega\, vau + \int_{\partial\Omega} ds\, vqu = \int_{\partial\Omega} ds\, vg + \int_\Omega d\Omega\, vf \tag{36}$$

which has to hold for all test functions v. (36) is the weak form of (32) with (33).

Galerkin[9] suggested to replace the notion of all possible solutions u and all test functions v by $u, v \in \mathcal{L}_n$, an n-dimensional linear space of suitable functions.

Let us select a base $\phi_1, \phi_2 \ldots, \phi_n$ for $\mathcal{L}_n$. (36) has to valid for $v = \phi_1, \phi_2, \ldots \phi_n$. We expand

$$u(x, y) = \sum_{j=1}^{n} U_j \phi_j(x, y) \tag{37}$$

and obtain a system of linear equations:

$$\sum_{j=1}^{n} K_{ij} U_j = F_i \quad . \tag{38}$$

The matrix K is given by

$$K_{ij} = \int_\Omega d\Omega \, (\boldsymbol{\nabla} \phi_i) \, c \, (\boldsymbol{\nabla} \phi_j) + \int_\Omega d\Omega \, \phi_i \, a \, \phi_j + \int_{\partial\Omega} ds \phi_i \, q \, \phi_j \quad , \tag{39}$$

the right hand side F by

$$F_i = \int_\Omega d\Omega \, \phi_i \, f + \int_{\partial\Omega} ds \, \phi_i \, g \quad . \tag{40}$$

In many applications we have to do with $c > 0$, $a \geq 0$, and $q \geq 0$. In this case

$$\|w\|^2 = \int_\Omega d\Omega \, \left\{ c |\boldsymbol{\nabla} w|^2 + a |w|^2 \right\} + \int_{\partial\Omega} ds \, q |w|^2 \tag{41}$$

describes a norm. The Galerkin approximation is optimal with respect to this norm.

2.2 Triangulation and tent functions

The region Ω will be covered by non-overlapping simplexes. These are intervals in one dimension, triangles in two dimensions, tetrahedrons in three dimensions, and so forth. The regions is made up of such simple finite elements.

Triangulation or the analogous procedure in three and more dimension is a subtle problem, we rely on the MATLAB PDE Toolbox. As an example we show in Figure **??** a triangulation for the heat conduction problem as discussed earlier.

[9] Boris Grigorievich Galerkin (1871-1945), Russian mathematician, inventor of the finite element method.

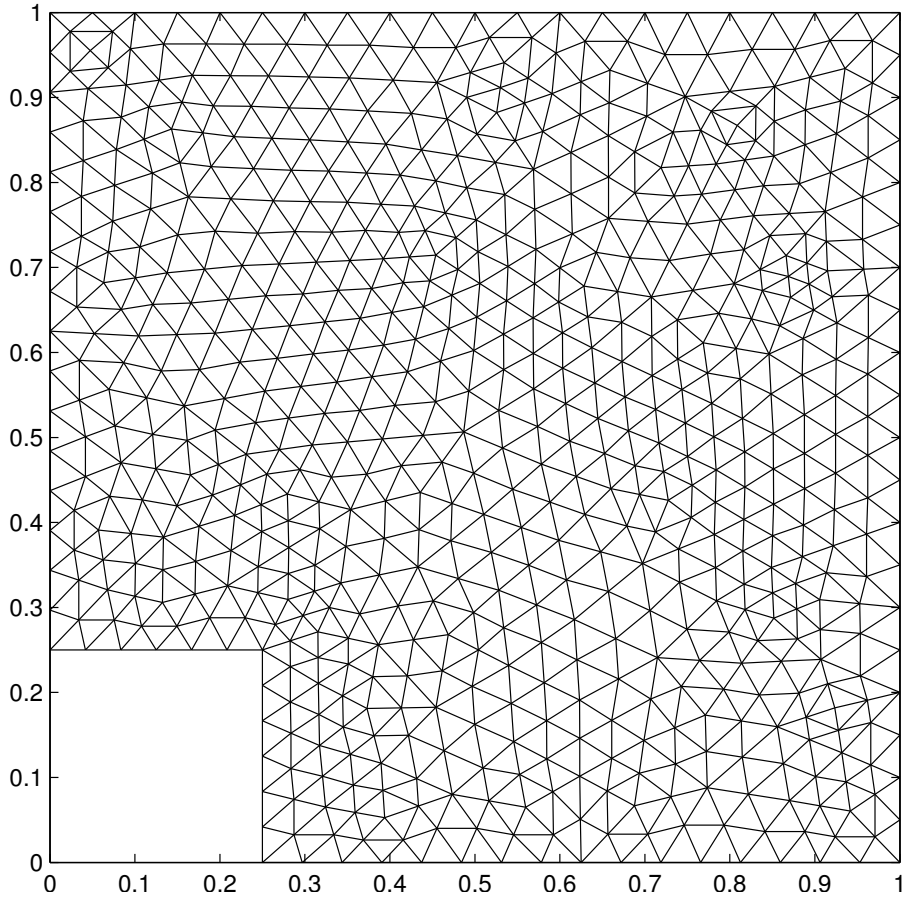

Figure 11: Triangulation of the region Ω between two parallel and concentric squares. Symmetry arguments allow to take only one quadrant into account. Von Neumann as well as Dirichlet boundary conditions have to be specified.

There are triangles, nodes, and edges. The boundary is a set of edges. We denote nodes by their coordinates, $p_i = (x_i, y_i)$. For each node i we construct a tent function ϕ_i as follows. ϕ_i is piecewise continuous and linear, and we demand

$$\phi_i(x_j, y_j) = \delta_{ij} \quad . \tag{42}$$

This choice has an important consequence: the expansion coefficients U_j of (37) are the field values at the corresponding nodes,

$$u(x_j, y_j) = U_j \quad . \tag{43}$$

Let us have a closer look at the triangulation as sketched in Figure **??**.

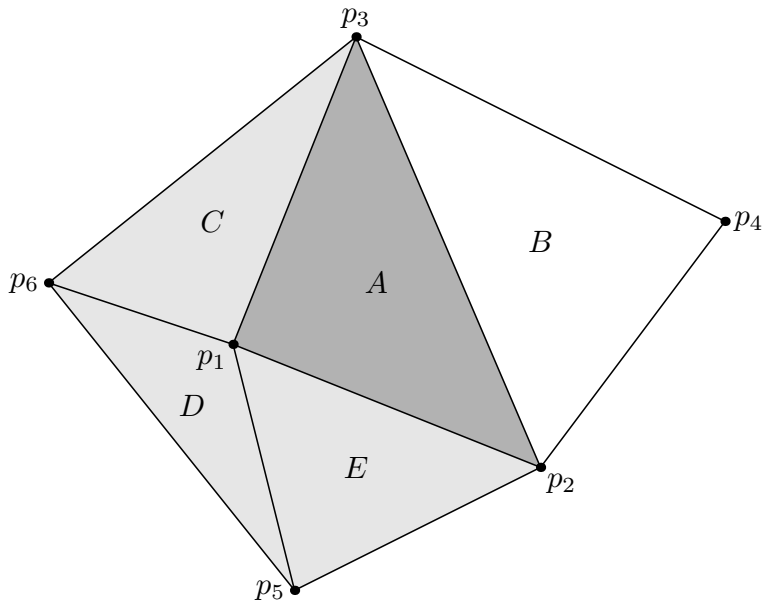

Figure 12: Neighborhood of node p_1. The tent function ϕ_1 has the value 1 at p_1 and falls off linearly towards the edges $p_3 - p_6$, $p_6 - p_5$, $p_5 - p_2$, and $p_2 - p_3$. It vanishes outside. Emphasis is on triangle A.

Tent function ϕ_1 does not vanish within triangles A, C, D, and E only, where it falls off linearly from 1 at p_1 towards the edges $p_3 - p_6$, $p_6 - p_5$, $p_5 - p_2$, and $p_2 - p_3$.
We here discuss ϕ_1 on triangle A. The area of A is Δ where

$$2\Delta = x_1y_2 + x_2y_3 + x_3y_1 - x_1y_3 - x_2y_1 - x_3y_2 \quad . \tag{44}$$

(44) is the result of rewriting $(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y1)$ or a cyclic permutation thereof. We define

$$s(x, y) = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{2\Delta} \quad . \tag{45}$$

s is linear in x and y, it vanishes at p_2 and p_3, and $s(x_1, y_1) = 1$. Therefore $\phi_1 = s$ on triangle A. We simply change indices and arrive at similar expressions for ϕ_1 on triangles C, D and E. We likewise construct expressions for ϕ_2 and ϕ_3 on triangle A. All remaining tent functions vanish on A.
The partial derivatives of ϕ_1 on A are given by

$$\partial_x\phi_1 = \frac{y_2 - y_3}{2\Delta} \quad \text{and} \quad \partial_y\phi_1 = \frac{x_3 - x_2}{2\Delta} \quad . \tag{46}$$

Hence the finite element A contributes to K_{11} by

$$\int_A d\Omega\, \phi_1 c \phi_1 = \bar{c}\,\Delta \left\{ \left(\frac{y_2 - y_3}{2\Delta} \right)^2 + \left(\frac{x_3 - x_2}{2\Delta} \right)^2 \right\} \quad . \tag{47}$$

Here $\bar{c}$ is the function $c = c(x, y)$ of (32) evaluated at the center of triangle A. The contribution to K_{12} is

$$\int_A d\Omega\, \phi_1 c \phi_2 = \bar{c}\,\Delta \left\{ \frac{y_2 - y_3}{2\Delta} \frac{y_3 - y_1}{2\Delta} + \frac{x_3 - x_2}{2\Delta} \frac{x_1 - x_3}{2\Delta} \right\} \quad . \tag{48}$$

These examples should suffice to explain how to assemble the various integrals:

- functions c, a, and f will be evaluated at the center of the actual triangle
- functions q and g are evaluated at the center of the actual edge
- partial derivatives are given by (46).

Triangle A for example with nodes p_1, p_2, and p_3 contributes to K_{11}, K_{12}, K_{13}, K_{21}, K_{22},K_{23}, K_{13}, K_{23},K_{33} as well as F_1, F_2, and F_3. On the other hand, contributions to K_{11} come from triangles A, C, D, and E.

2.3 A worked out example

We refer to the heat conduction problem of subsection 1.3. This time we do not solve it by the finite difference method but with the MATLAB PDE Toolbox. This toolbox implements the finite element method for solving a large class of partial differential equations[10].

```
>> pdetool
```

at the MATLAB prompt opens a graphical user interface for specifying and solving a partial differential equation. You may compose Ω out of rectangles (including squares), ellipses (including circles) and polygons. These primitive regions can be combined by the union, intersection and difference operations. Once Ω has been constructed, you may go to boundary mode. By clicking on one ore more boundary sections, the boundary conditions may be specified. Next you specify the partial differential equation to be solved. Thereafter you create an initial mesh of triangles which should be refined at least once. Now one may solve the problem and plot the solution. The mesh (`[p,e,t]`)[11] and the solution `u` may be exported to the MATLAB workspace,

[10] Note that MATLAB identifies PDE (partial differential equations) with the finite element method FEM to solve them.

[11] points, edges of the boundary, and triangles

and th Symmetry arguments allow to take only one quadrant into account. e solver function as well. See Figure 11 for the triangularization which we have employed.

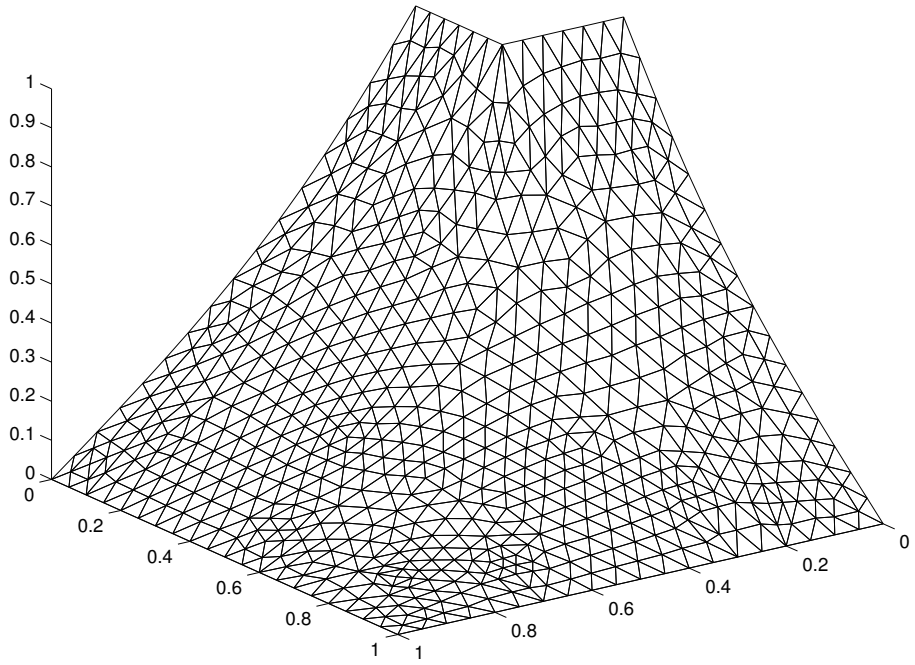

Figure 13: FEM solution of $u_{xx} + u_{yy} = 0$ with $u = 1$ on the inner and $u = 0$ on the outer square. The triangulation shown in Figure 11 has been used. For symmetry reasons, only one quarter of the region Ω was investigated. Therefor von Neumann as well as Dirichlet boundary conditions had to be implemented.

The MATLAB PDE Toolbox is not restricted to domains Ω which are made up of rectangles and circles. You may specify a file `problemg.m` which describes the geometry of your problem. Inspect `lshapeg.m` as an example.
The MATLAB PDE Toolbox provides not only for elliptic, but also for parabolic, hyperbolic and eigenvalue problems. Prototypes are

- $u_{xx} + u_{yy} = 0$
- $u_t - u_{xx} = 0$
- $u_{tt} - u_{xx} = 0$
- $u_{xx} + u_{yy} - \lambda u = 0$

3 Propagation

The heat equation

$$\dot{T} = \kappa \Delta T \tag{49}$$

for the temperature field $T = T(t, \boldsymbol{x})$ is the prototype of an initial value problem. For time $t = 0$ the temperature distribution $T(0, \boldsymbol{x}) = T_0(\boldsymbol{x})$ is given, and we want to find $T(t, \boldsymbol{x})$ for $t > 0$. So far we have studied the final, or stationary state only which is described by $\dot{T} = 0$, or $\Delta T = 0$. By the way, it is always advisable to chose proper units of space and time such that (49) reads

$$u_t = u_{xx} + u_{yy} + u_{zz} \quad . \tag{50}$$

The Fresnel[12] equation is another prototype example. Monochromatic light in an isotropic medium with smoothly varying permittivity $\epsilon = \epsilon(x, y, z)$ is described by the Helmholtz equation

$$\Delta E = k_0^2 \epsilon E \quad , \tag{51}$$

where E is any component of the electric field and $k_0 = \omega/c = 2\pi/\lambda$ is the vacuum wave number. We now assume that u is almost a plain wave,

$$E(x, y, z) = u(x, y, z)\, e^{\,i\beta z} \tag{52}$$

such that

$$|u_{zz}| \ll k_0 |u_z| \tag{53}$$

holds true. We then obtain

$$iu_z = \frac{-u_{xx} - u_{yy} + (\beta^2 - k_0^2 \epsilon)\, u}{2\beta} \quad , \tag{54}$$

the Fresnel equation. Again, $u(x, y, 0) = u_0(x, y)$ is prescribed, and we want to know the field $u(x, y, z)$ at z. Note that the stationary case $iu_z = 0$ describes modes which we have discussed before.
Again, this equation may be written as

$$-iu_z = u_{xx} + u_{yy} + \eta u \tag{55}$$

[12] Augustin-Jean Fresnel (1788-1827), French physicist, pioneer of wave optics. He maintained that 'not even acclaim from distinguished colleagues could compare with the pleasure of discovering a theoretical truth or confirming a calculation experimentally'.

by choosing proper units.
The initial value problem is solved most often by a finite difference method: advance from $u(t, \boldsymbol{x})$ to $u(t+h, \boldsymbol{x})$ where t should be replaced by z in case of the Fresnel equation. There are various possibilities, but only some of them are stable. The Crank-Nicholson scheme is stable for both prototype propagation problems, we study it in detail. We also discuss the classic Feit and Fleck beam propagation algorithm.

3.1 Stability considerations

Von Neumann has studied the stability of various propagation algorithms. Both prototypes, the heat equation and the Fresnel equation, are of first order with respect to the propagation coordinate (t or z) and of second order with respect to the cross section coordinates. To keep thing simple, we allow for one space coordinate x only (besides the propagation coordinate.
We represent the field $u = u(t, x)$ or $u = u(z, x)$ on a mesh:

$$u_r^n = u(n\tau, rh) \text{ where } n, r \in \mathbb{Z} \ . \tag{56}$$

When the propagation step size τ and the cross section spacing h are choosen very, very small, the coefficients in front of the partial derivative operators become constant, and multiplication terms even drop out. Locally, there is a polynomial of differentiation operators which is best dealt with by a Fourier decomposition. So, for a particular space frequency k we may write

$$u_r^n = \xi(k)^n \, e^{ikrh} \ . \tag{57}$$

The propagation scheme $u_r^n \to u_r^{n+1}$ is stable if $|\xi(k)| \leq 1$ for all real space frequencies k.

Explicit forward

Naively you might write for the heat equation $u_t = u_{xx}$

$$\frac{u_r^{n+1} - u_r^n}{\tau} = \frac{u_{r+1}^n - 2u_r^n + u_{r-1}^n}{h^2} \ . \tag{58}$$

We call this scheme 'forward in time, centered in space'. Inserting (57) yields

$$\xi(k) = 1 - \frac{4\tau}{h^2} \sin^2 \frac{kh}{2} \ . \tag{59}$$

For the Fresnel equation we obtain

$$\xi(k) = 1 - i\frac{4\tau}{h^2} \sin^2 \frac{kh}{2} \ . \tag{60}$$

'Forward in time, centered in space' is stable for the heat equation provided $2\tau \leq h^2$. It is never stable for the Fresnel equation.

Implicit forward

One might employ the above propagation scheme backward. The new coefficients u_r^{n+1} are chosen such that the values u_r^n result. This scheme is therefore implicit and termed 'backward in time, centered in space'. We set

$$\frac{u_r^{n+1} - u_r^n}{\tau} = \frac{u_{r+1}^{n+1} - 2u_r^{n+1} + u_{r-1}^{n+1}}{h^2} \tag{61}$$

and obtain

$$\frac{1}{\xi(k)} = 1 + \frac{4\tau}{h^2} \sin^2 \frac{kh}{2} \tag{62}$$

for the heat equation and

$$\frac{1}{\xi(k)} = 1 + i\frac{4\tau}{h^2} \sin^2 \frac{kh}{2} \tag{63}$$

for the Fresnel equation. In both cases the propagation scheme is stable.

Averaging explicit and implicit forward

Both schemes presented so far are biased with respect to the propagation direction. Therefore, it is an appealing idea to combine them symmetrically. We propagate forward from $n\tau$ by $\tau/2$ and backward from $(n+1)\tau$ by $-\tau/2$ and demand that the results are equal:

$$u_r^n + \frac{\tau}{2}\frac{u_{r+1}^n - 2u_r^n + u_{r-1}^n}{h^2} = u_r^{n+1} - \frac{\tau}{2}\frac{u_{r+1}^{n+1} - 2u_r^{n+1} + u_{r-1}^{n+1}}{h^2} \quad . \tag{64}$$

The heat equations gives rise to

$$\xi(k) = \frac{1 - (2\tau/h^2)\sin^2(kh/2)}{1 + (2\tau/h^2)\sin^2(kh/2)} \quad , \tag{65}$$

the Fresnel equation to

$$\xi(k) = \frac{1 - i(2\tau/h^2)\sin^2(kh/2)}{1 + i(2\tau/h^2)\sin^2(kh/2)} \quad . \tag{66}$$

This propagation scheme, which was suggested by Crank and Nicholson, is always stable. Moreover, it is correct to one more order of the time step τ.

3.2 Crank-Nicholson scheme

As an example we will implement the Crank-Nicholson scheme for the propagation of light as described by the Fresnel equation. We restrict ourselves to a one-dimensional cross section. We have to solve

$$u_z = \frac{i}{2\beta}\left\{u_{xx} + k_0^2\epsilon - \beta^2\right\} \tag{67}$$

where $u(x,0) = u_0(x)$ is prescribed.

Homogeneous medium

Recall that β is a reference propagation constant which should be chosen such that the envelop function u in (52) depends but slowly on z. Here we discuss a homogeneous medium, the permittivity profile $\epsilon(x)$ does not depend on x. We therefore choose $\beta = k_0\sqrt{\epsilon}$ so that

$$u_z = \frac{i}{2\beta}u_{xx} \tag{68}$$

has to be solved.

Let us first setup the wavelength (He-Ne laser), the refractive index (glass), and the computational window. All lengths are in microns.

```
1   % this file is propag_hm.m
2   lambda=0.633;
3   k0=2*pi/lambda;
4   rfr_ndx=1.50;
5   beta=k0*rfr_ndx;
6   hx=0.1;
7   wx=10.0;
8   x=[-0.5*wx:hx:0.5*wx]';
9   N=size(x,1);
```

Now we formulate the propagation scheme. Stepping forward $h_z/2$ is achieved by matrix F, stepping backward by matrix B. d_m and d_n are the main and next to main diagonals. We have implemented (59) for the Fresnel equation by inserting i at the proper place and dividing by 2β.

```
10  hz=1.0;
11  f=0.5*i*hz/(2*beta);
12  dm=f*(-2*ones(N,1))/hx^2;
13  dn=f*ones(N-1,1)/hx^2;
14  F=eye(N)+diag(dn,-1)+diag(dm,0)+diag(dn,1);
15  B=eye(N)-diag(dn,-1)-diag(dm,0)-diag(dn,1);
```

It remains to step forward by solving $u \to B^{-1}Fu$. The intensity history is recorded and plotted.

```
16   Z=60;
17   history=zeros(N,Z);
18   width=1.0;
19   u=exp(-(x/width).^2);
20   for n=1:Z
21      history(:,n)=abs(u).^2;
22      u=B\(F*u);
23   end
24   mesh(1:Z,x,history)
```

The result is plotted as Figure 14.

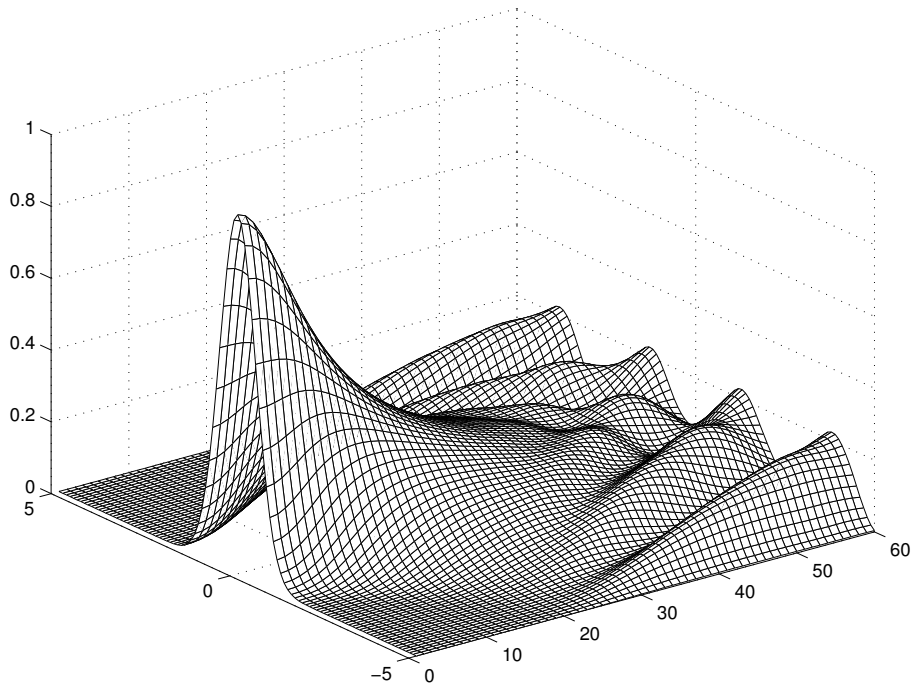

Figure 14: A Gaussian beam enters from left and propagates in a homogeneous medium. The light intensity is plotted vs. x for 60 propagation steps. The computational window is 10 μm wide, propagation is in steps of 1.0 μm.

What has gone wrong? Nothing initially, the Gaussian beams spreads out as it should until it reaches the edges of the computational window. In fact, we silently assumed that the field vanishes outside the computational window which amounts to reflection at the window boundaries. Figure 14

correctly describes the propagation of light between two parallel mirrors, with beautiful interference effects.

Transparent boundary conditions

However, we wanted to study the propagation of light in a transparent medium. Headly[13] has suggested a method for simulating a transparent computational window. He assumes, and enforces, the field to be outgoing waves at the boundaries.

Let us discuss the upper end $x = W/2$. If $u \propto e^{ikx}$, and if the real part of k is positive, then the waves flow outward, as they should. If not, the boundary conditions are manipulated in such a way that the real part of k vanishes.

Assume the points within the window to be labeled by $1, 2, \ldots, N$. From the old field u^n we work out the wave number k by setting

$$e^{ikh_x} = \frac{u_N^n}{u_{N-1}^n} \quad . \tag{69}$$

If the real part of k should turn out to be negative, we change it to zero. With the possibly modified wave number $\bar{k}$ we define

$$\gamma = e^{i\bar{k}h_x} \quad , \tag{70}$$

and we program as if

$$u_{N+1}^n = \gamma u_N^n \ \text{ and } \ u_{N+1}^{n+1} = \gamma u_N^{n+1} \tag{71}$$

would hold, thereby enforcing outgoing waves at the boundary.

An analogous procedure is applied to the lower end of the computational window.

The new program is the same up to the propagation statement:

```
1  % this file is propag_tb.m
2  TINY=1.0e-4;
3  lambda=0.633;
4  k0=2*pi/lambda;
5  rfr_ndx=1.50;
6  beta=k0*rfr_ndx;
7  hx=0.1;
8  wx=10.0;
```

[13] G. R. Headley, Transparent boundary conditions for the beam propagation method, IEEE Journal of Quantum Electronics 28 (1992) 363-370

```
 9   x=[-0.5*wx:hx:0.5*wx]';
10   N=size(x,1);
11   hz=1.0;
12   f=0.5*i*hz/(2*beta);
13   dm=f*(-2*ones(N,1))/hx^2;
14   dn=f*ones(N-1,1)/hx^2;
15   F=eye(N)+diag(dn,-1)+diag(dm,0)+diag(dn,1);
16   B=eye(N)-diag(dn,-1)-diag(dm,0)-diag(dn,1);
17   Z=60;
18   history=zeros(N,Z);
19   width=1.0;
20   u=exp(-(x/width).^2);
21   for n=1:Z
22      history(:,n)=abs(u).^2;
```

At this point we had written `u=B\(F*u);`, which proved to be too simple. We modify:

```
23      FF=F;
24      BB=B;
25      if abs(u(1))>TINY
26         k=i/hx*log(u(2)/u(1));
27         if real(k)<0 k=i*imag(k); end
28         tbc=exp(i*k*hx)*f/hx^2;
29         FF(1,1)=FF(1,1)+tbc;
30         BB(1,1)=BB(1,1)-tbc;
31      end
32      if abs(u(N))>TINY
33         k=-i/hx*log(u(N)/u(N-1));
34         if real(k)<0 k=i*imag(k); end
35         tbc=exp(i*k*hx)*f/hx^2;
36         FF(N,N)=FF(N,N)+tbc;
37         BB(N,N)=BB(N,N)-tbc;
38      end
39      u=BB\(FF*u);
```

The rest remains,

```
40   end;
41   mesh(1:Z,x,history)
```

And indeed, the Gaussian gets wider and wider, but there is no reflection from the boundaries of the artificial computational window which is convincingly demonstrated in Figure 15.

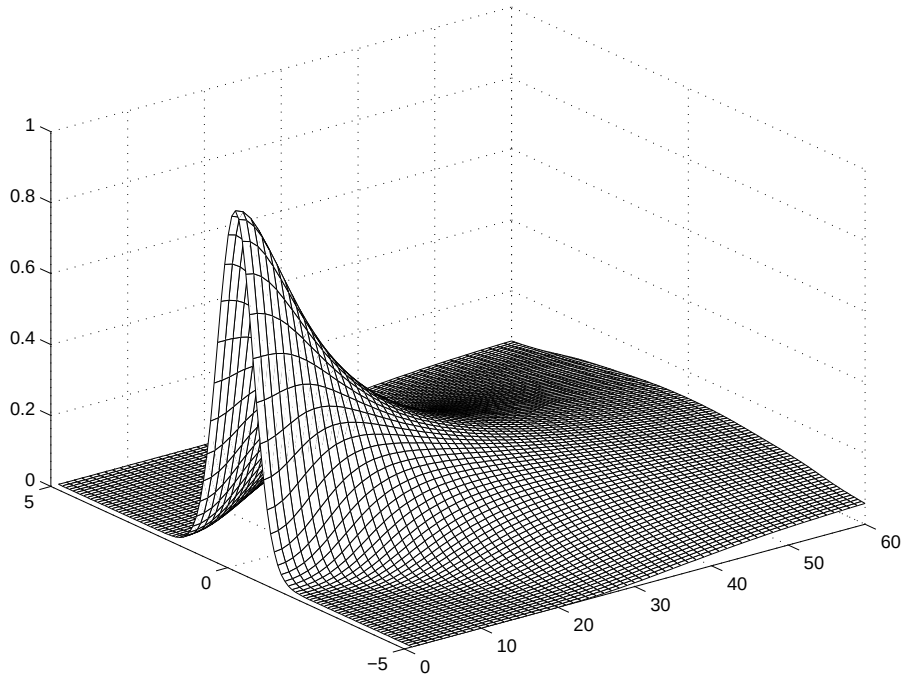

Figure 15: Same as Figure 14, but this time transparent boundary conditions have been implemented.

Propagation in a waveguide

Assume there is a slab of thickness 3.0 μm the refractive index of which is slightly higher. Let us increase the permittivity of the slab by 0.025, for example.

We add one line of code,

```
del_eps=0.025*(abs(x)<=1.5);
```

and change the definition of the main diagonal into

```
dm=f*(-2*ones(N,1)/hx^2+k0^2*del_eps);
```

Moreover, we propagate over a longer distance, `Z=300`.

The entire program listing can be found in the appendix.

With these slight modifications we have simulated how a guided mode is excited by a Gaussian beam, as shown in Figure 16

We have plotted the power within the computational window by the following simple program:

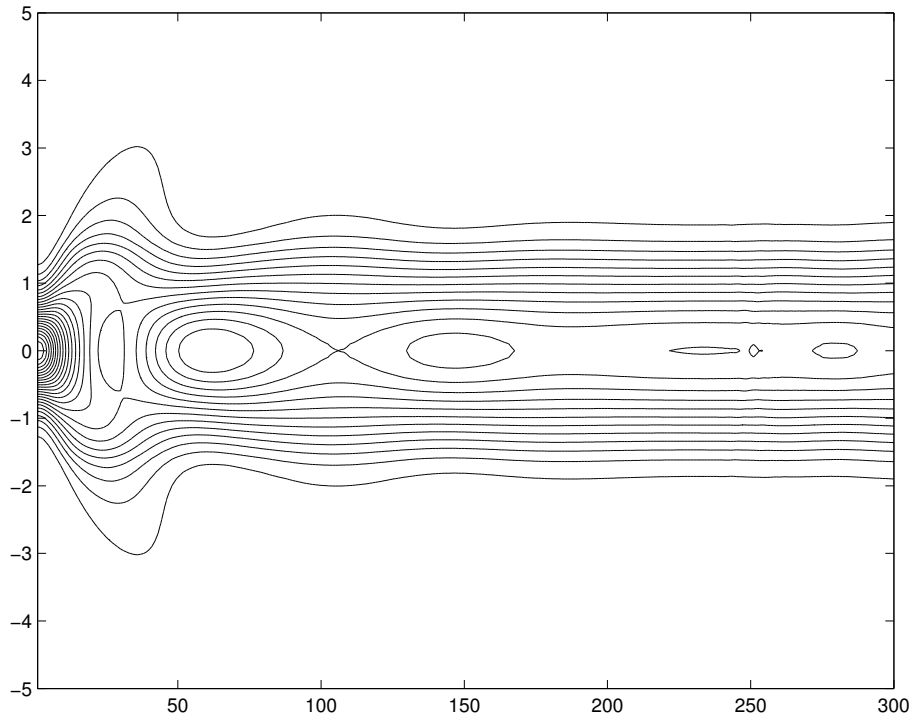

Figure 16: The propagation of an initially Gaussian beam in a slab waveguide. Contour lines refer to the light intensity. Part of the light is radiated away, the rest finally propagates as a guided wave.

```
1   pw=sum(history,1);
2   plot(pw/pw(1));
3   axis([0,Z*hz,0.8,1]);
```

Let us recall the dimensions: the computational window is 10 μm wide, the guiding layer has a thickness of 3.0 μm. It takes more than 200 μm until the mode has formed. This is no problem for a glass fiber, but intolerable for an integrated optical device. The beam profile must be shaped smoothly by a taper in order to avoid insertion losses and reduce mode formation lengths.

It is left as an exercise to the reader to play with parameters. What happens if the Gaussian beam and the guided mode have a better overlap? What happens if the waveguide is excited off-center? Make the waveguide narrower, the permittivity of the guiding slab higher, and so forth.

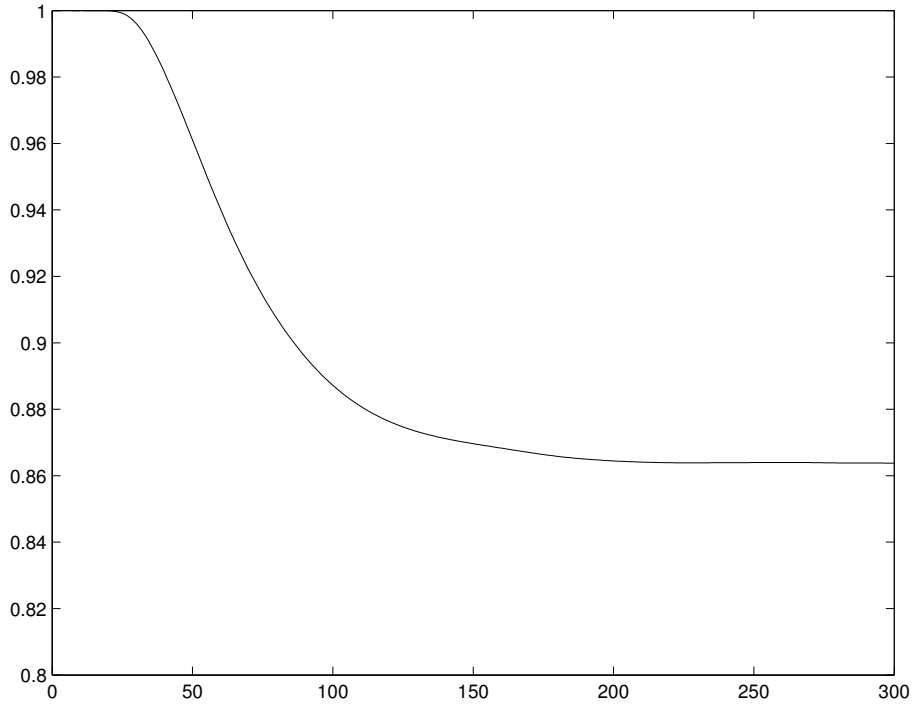

Figure 17: Normalized power within the computational window vs. propagation distance (microns). 86 % of the initial power finally propagates without further radiation losses. Note that it takes a certain distance until the incoming light has reached the border of the computational window.

A Matlab

MATLAB is concerned with matrices of numbers. Natural, integer, rational, real, and complex numbers are treated alike, as implied by $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$. An $m \times n$ matrix consists of m rows of length n, or of n columns of length m.

Generating matrices

A 2×3 matrix like

$$a = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

has two rows and three columns. Here a column vector is a 2×1 matrix, a row vector is a 1×3 matrix.
Matrices are constructed within square brackets [...]. Submatrices (in particular numbers) are concatenated horizontally by the `,` operator and vertically by the `;` operator. On the MATLAB prompt we may write

```
>> a=[1,2,3;4,5,6]
```

or

```
>> a=[[1,2,3];[4,5,6]]
```

or

```
>> a=[[1;4],[2;5],[3;6]]
```

or

```
>> a=[[[1,2];[4,5]],[3;6]]
```

Another possibility is

```
>> r1=[1,2,3]
>> r2=[4,5,6]
>> a=[r1;r2]
```

Likewise

```
>> c1=[1;4]
>> c2=[2;5]
>> c3=[3;6]
>> a=[c1,c2,c3]
```

Instead of inserting numbers one after the other, one may generate equally spaced sequences by the `from:step:to` construct. `[0:0.1:10]` is an example, a row vector with 101 elements. `x:y` is short for `x:1:y`.
The ' operator effects transposing and complex conjugating. As an example,

```
>> row=[1,2+3i]
>> col=row'
```

is the same as

```
>> col=[1;2-3i]
```

Note the letter i for the imaginary unit.
`zeros(m,n)` generates an $m \times n$ matrix of zeros. `ones(m,n)` works likewise. `eye(m)` creates an $m \times m$ (square) unit matrix[14]. `rand(m,n)` produces an $m \times n$ matrix of random numbers which are uniformly distributed in $[0, 1]$. `randn` generates normally distributed random numbers with mean 0 and variance 1.
Adding by the + operator, subtracting by the - operator and multiplying matrices by the * operator works as expected. It is an error if the matrices to be combined do not match. Matrices of equal shape may be multiplied or divided elementwise by the .* or ./ operators. Most functions, like `sin`, `sqrt` etc. operate elementwise on matrices. Try `sqrt([0:2]'*[0:4])`.

Displaying matrices

The result of an assignment like `a=rand(2,2)` is automatically displayed. Numbers, vectors and matrices are sent to the MATLAB command window. The output `format` command (see `>> help format`) allows to toggle between legibility and accuracy. Output to the MATLAB window is stopped if the command is terminated by a semicolon.
In many cases, a vector represents the values of a variable which should be displayed graphically.
In its simplest form the `plot` command plots a vector vs. index. Try the following command sequence:

```
>> x=pi*[0:0.1:2];
>> y=sin(x);
>> plot(y)
```

[14]The symbols I and i are already reserved for the imaginary unit. The unity matrix is therefore denoted by eye which is pronounced in English as I.

However, `plot(x,y)` is better. Note that we delimited the first two commands by a semicolon thus suppressing output to the MATLAB command window.

Saving and loading matrices

The following two lines produce a matrix `a` and save it to a file:

```
>> a=rand(8,4);
>> save test.dat a -ascii
```

The corresponding load command is

```
>> load test.dat -ascii
```

It produces a matrix `test` which (almost) reproduces the original matrix `a`. Almost, because data are written in legible ascii format with eight digits precision. If this is not sufficient, write `-ascii -double` instead. Without the ascii modifier, data would be written in binary format, and such binary data files are useless for any other program than MATLAB.
There are also low-level file input/output operations which are useful if data come from or are intended for another program.

Accessing matrix elements

The elements of a matrix are addressed by specifying a (row,column) index pair, such as `a(2,3)` for a_{23}. The result is a number.
Instead of an index, one may also specify a vector of indices. Thus, if `a` is the above 2×3 matrix, the assignment `b=a([1,2],[1,3])` results in

$$b = \begin{pmatrix} 1 & 3 \\ 4 & 6 \end{pmatrix}$$

The double colon `:` is short for all indices. We could also write `b=a(:,[1,3])`. `a(m,:)` is the m'th row and `a(:,n)` the n'th column. `a(:,:)` is `a` itself.
The dimensions of a matrix `a` are determined by `[m,n]=size(a)`. Thus, erasing the first row and the first column may be programmed as

```
>> [rows,cols]=size(a);
>> b=a([2:rows],[2:cols]);
```

There are also functions for extracting diagonals (`diag`) and lower or upper triangular parts (`tril` and `triu`). Creating a symmetric matrix S of normally distributed random numbers may be achieved by

```
a=randn(dim);
b=triu(a,0)+triu(a,1)';
```

Solving linear equations

Solving `a*x=y` for a square matrix `a` and a column vector `y` is surprisingly simple. Just say

```
>> x=a\y;
```

This command has the same effect as `x=inv(a)*y` where `inv(a)` is the inverse a^{-1} of a. The division operator $\boxed{\backslash}$ however is much faster. After all, `inv(a)` solves all linear equations with a, not only one.
The following piece of code is self-explaining:

```
>> a=randn(5);   % an arbitrary 5 by 5 matrix
>> y=randn(5,1); % an arbitrary column vector
>> x=a\y;        % solve a*x=y
>> norm(y-a*x)   % y-a*x should be zero
```

Note that text following a percent sign is ignored (comment).
For completeness: there is also a left division operator $\boxed{/}$. It solves equations like `x*a=y` for row vectors `y` by `x=y/a` which is a row vector as well.

Diagonalization

Recall that a square matrix A (not necessarily hermitian) may be written as $AV = VD$ where D is diagonal. The diagonal elements $d^{(k)} = D_{kk}$ are the eigenvalues of A, the column vectors $v_i^{(k)} = V_{ik}$ the corresponding eigenvectors. The `eig` command comes in two versions. `d=eig(A)` computes only the eigenvalues of `A`. `[V,D]=eig(A)` computes the matrix `V` of eigenvectors as well as the diagonal matrix `D` of eigenvalues. Try this:

```
>> A=randn(5);
>> d=eig(A);
>> [V,D]=eig(A);
>> norm(d-diag(D))
>> v1=V(:,1);
>> d1=d(1);
>> norm(A*v1-d1*v1)
```

By the way, such a long sequence of instructions should be written line by line to a script file (without the MATLAB prompt `>>`). Call this file `test_eig.m`, the extension indicating MATLAB. You may now issue the command

```
>> test_eig
```

MATLAB searches for a file `test_eig.m` and executes its instructions.

Ordinary differential equations

Ordinary differential equations can be transformed into systems of first order differential equations. The state $\boldsymbol{x}$, an n-dimensional vector, depends on a real parameter t. The system of first order differential equations is described by the slope field $\boldsymbol{f} = \boldsymbol{f}(t, \boldsymbol{x})$ such that

$$\frac{dx_i(t)}{dt} = f_i(t, x_1(t), x_2(t), \ldots, x_n(t))$$

holds for all t.
Before we can solve a system of ordinary differential equations, we must describe it. We have to specify a function, the input being a scalar t and a vector x, the output is $\dot{\boldsymbol{x}}$.
Let us come back to the Kepler problem of planetary motion. The planet moves in a plane. At a certain time t it is located at x_1, x_2 and has a velocity $x_3 = v_1 = \dot{x}_1, x_4 = v_2 = \dot{x}_2$. In suitable units we have to solve

$$\dot{x}_1 = x_3 \ ; \ \dot{x}_2 = x_4 \ ; \ \dot{x}_3 = -x_1/r^3 \ \text{ and } \ \dot{x}_4 = -x_2/r^3$$

where $r = \sqrt{x_1^2 + x_2^2}$.
You should create a file `kepler.m` with the following lines of text:

```
function xdot = kepler(t, x)
  r3=(x(1)^2+x(2)^2)^(3/2);
  xdot=[x(3);x(4);-x(1)/r3;-x(2)/r3];
```

This slope field describes the system of ordinary differential equations (ODE) to be solved.
The standard MATLAB program is `ode45`. It requires a function (here: `kepler` in file `kepler.m`), a vector of times for which a state is to be calculated, and an initial state. We will start the planet at $(1, 0)$ with initial velocity $(0, 0.8)$. We also set the desired relative accuracy.

```
>> options=odeset('RelTol',1e-6);
>> [t,x]=ode45('kepler',[0:0.1:10],[1,0,0,0.8],options);
>> x1=x(:,1); x2=x(:,2);
>> plot(x1,x2);
```

One may check the accuracy of this solution by working out the total energy:

```
>> v1=x(:,3); v2=x(:,4);
>> te=(v1.^2+v2.^2)/2-1./sqrt(x1.^2+x2.^2);
>> max(te)-min(te)
```

Consult the documentation for more ODE solvers and for control options.

...and much more

- MATLAB knows many special functions (Airy, Bessel, Beta, etc.).
- `fminsearch` finds the minimum of a real valued function of many arguments (simplex method of Nelder and Mead).
- `fft` performs the fast Fourier transform of vectors or matrices (columnwise).
- `polyfit` fits data to a polynomial.
- MATLAB can work with sparse matrices. This is particularly import for partial differential equations. Call `>> help sparfun` for a summary of ML's sparse matrix capabilities.
- There is a rich set of graphic functions, 2D and 3D as well.

Avoid loops!

After a while you think about `x=linspace(-1,1,1024)` as a variable. Of which your need the square, `x2=x.^2`, for example. Such block operations are fast in MATLAB. Loops (repetitive commands) however are interpreted, they should be avoided. Just look at the following example where we have multiplied two matrices, first by a matrix command, then by looping:

```
% this file is test_loop.m
% compare matrix and loop operations
dim=300;
a=randn(dim);
b=randn(dim);
tic;
c=a*b;
toc;
tic;
c=zeros(dim,dim);
for i=1:dim
   for k=1:dim
      sum=0;
      for j=1:dim
```

```
         sum=sum+a(i,j)*b(j,k);
      end;
      c(i,k)=sum;
   end;
end;
toc;
```

On my laptop (2008) the cpu-times were 0.020 s and 110 s. The factor 5,000 in execution time stresses what I said before: avoid loops whenever possible, in particular nested loops.

B Program listing

The following program has produced Figure 16.

```
1   TINY=1.0e-4;
2   lambda=0.633;
3   k0=2*pi/lambda;
4   rfr_ndx=1.50;
5   beta=k0*rfr_ndx;
6   hx=0.1;
7   wx=10.0;
8   x=[-0.5*wx:hx:0.5*wx]';
9   N=size(x,1);
10  del_eps=0.025*(abs(x)<=1.5);
11  hz=1.0;
12  f=0.5*i*hz/(2*beta);
13  dm=f*(-2*ones(N,1)/hx^2+k0^2*del_eps);
14  dn=f*ones(N-1,1)/hx^2;
15  F=eye(N)+diag(dn,-1)+diag(dm,0)+diag(dn,1);
16  B=eye(N)-diag(dn,-1)-diag(dm,0)-diag(dn,1);
17  Z=300;
18  history=zeros(N,Z);
19  u=exp(-(x/1.0).^2);
20  for n=1:Z
21     history(:,n)=abs(u).^2;
22     FF=F;
23     BB=B;
24     if abs(u(1))>TINY
25        k=i/hx*log(u(2)/u(1));
26        if real(k)<0 k=imag(k); end
27        tbc=exp(i*k*hx)*f/hx^2;
28        FF(1,1)=FF(1,1)+tbc;
29        BB(1,1)=BB(1,1)-tbc;
30     end
31     if abs(u(N))>TINY
32        k=-i/hx*log(u(N)/u(N-1));
33        if real(k)<0 k=imag(k); end
34        tbc=exp(i*k*hx)*f/hx^2;
35        FF(N,N)=FF(N,N)+tbc;
36        BB(N,N)=BB(N,N)-tbc;
37     end
38     u=BB\(FF*u);
39  end
40  contour(1:Z,x,history,24,'k-');
```