Computational Physics

Peter Hertel

Fachbereich Physik
Universität Osnabrück

Numerical, or computational physics is a branch of physics with an age-old tradition. If you can't produce a number, you have achieved nothing says Feynman, and right he is. More often than not this is possible only by computing. This series of lectures introduces standard methods of computational physics. The computational power which is available to the average student or researcher has grown by a factor of thousand or so, within the last twenty years. And with it the availability of high quality software. Therefore, these lectures concentrate on how to use software, and not how to develop it. Quite naturally, this boils down to MATLAB.

January 14, 2008

Contents

1 Introduction

This series of lectures is about standard numerical techniques for computing in physics. Although teachers prefer examples which can be solved by pencil and paper, or by chalk on the blackboard, most problems cannot be solved analytically. They require computations. Computational physics has a long tradition. However, computational physics cannot be taught any more in the traditional way.

In 1980, the first IBM personal computer could address 64 kB of memory, its mass storage was a floppy of 360 kB, and it ran at 8 MHz with an 8 bit processor. With 15000 EUR it was cheap as compared to mainframes. Today, an electronic computer, such as my laptop, costs 1500 EUR. Its 32 bit processor runs at 1.2 GHz, is supported by 512 MB fast storage, 40 GB mass storage, ethernet and modem controller on board, and so forth. This amounts to a many-hundred-fold increase in computational power within the last two decades.

Moreover, high quality software to be run on these cheap and highly efficient computers has become cheap. Just study the price list of the MathWorks corporation for MATLAB.

Today, a course on computational physics, in particular an introduction, should concentrate on how to use available and cheap high quality software, and not how to develop it.

I have decided on using MATLAB for various reasons.

First, because it is relatively cheap.

Second, because it comes with an integrated development environment such as a command window, a language sensitive editor, a debugger, a workspace inspector, and so forth.

Third, because it includes the software treasures of the last 50 years. In particular, the `LINPACK` and `EISPACK` packages[1] of linear algebra, forming the backbone of computational physics, are the core of MATLAB. In fact, MATLAB was conceived as a meta-language how to use such `FORTRAN` code without indulging into unnecessary details.

Fourth, MATLAB now provides for sparse matrix technology which is essential for partial differential equations.

Fifth, MATLAB makes you think in terms of objects, it is not a procedural language. After a short while, you will think of x as a variable, not of an array of equally spaced representative values.

And last but not least, MATLAB is a powerful graphics machine for visual-

[1] Steve Moler is one of the authors of the the `LINPACK` and `EISPACK` packages documentation, and one of its main contributors. Today he is the chief executive officer of MathWorks.

izing data, both two and three dimensional. In particular, MATLAB allows for the export of pictures in encapsulated postscript format.

In these lectures on Computational Physics we cannot cover a number of other important fields, such as data acquisition, graphical user interfaces or fine points of graphics programming. Instead, we will restrict ourselves to standard techniques of doing analysis on a computer.

This text addresses students of physics who are used to learn by carefully chosen examples. It is neither systematic nor complete.

2 Matrices

The basic objects of MATLAB are matrices. A number is a `1` × `1` matrix. A row vector is `1` × `N` matrix, a column vector is an `N` × `1` matrix. A polynomial is represented by the row vector of its coefficients. A string is a row vector of 2 byte numbers which represent characters. And so forth.

The most important operators are square brackets for aggregation. The comma operator separates objects which are to be aggregated horizontally. The semicolon operator separates objects which are to be aggregated vertically. The semicolon operator, at the end of an expression, suppresses echoing.

Here are a few examples:

```
>> A=[1,2,3;4,5,6];
>> B=[A;[7,8,9]];
```

The `size` function takes in a matrix and returns the numbers of rows and columns. Try

```
>> [r,c]=size(A);
```

Note that output is always aggregated horizontally.

By saying[2]

```
>> M=3; N=5;
>> a=zeros(M,N);
```

you create an `M` × `N` matrix `a` of zeros. Likewise, `ones` creates matrices of ones.

```
>> B=eye(N);
```

[2] MATLAB distinguishes between lower and upper case

generates a square `5 × 5` unit matrix. Check it by typing

```
>> B(2,1)
>> B(2,2)
```

The dash operator stands for Hermitian conjugation. Compare

```
>> x=[1,2+3i,4;0,i,2]
```

and

```
>> xc=x'
```

One of the highlights of MATLAB is the colon operator. It stands for '`from:to`. The colon itself stands for all indices. Thus we could have written

```
>> B=[1:3;4:6;7:9];
```

We now extract the first and the third row:

```
>> C=B([1,3],:);
```

Note that matrices are used just as functions. Arguments are in parentheses and separated by commas. The first argument is a vector of column indices, the second argument a vector of row indices.

Another often used function for creating vectors is `linspace`. This function has three arguments: the lower value of an interval, the upper value, and the number of representative points.

```
>> x=linspace(0,10,128);
```

produces a row vector of size `1 × 128`. Its first element is 0, the last is 10, and linear interpolation in-between.

```
>> x=[0:10/127:10];
```

does the same.

Here the `from:step:to` construct is used. If the difference between `to` and `from` is not an integer multiple of `step`, then `to` is an upper bound. If `step` is omitted, a value of 1 is assumed.

From the `10 × 10` matrix

```
>> x=rand(10,10);
```

of randomly chosen numbers we may extract the sub-matrix

```
>> y=x(1:2:10,1:2:10);
```

of random numbers with odd indices.

Constructs like

```
>> k=[1,4,9:2:17,31];
```

are allowed as well. MATLAB's syntax rules, if strictly applied, would require

```
>> k=[[1],[4],[9:2:17],[31]];
```

They are, however, sensibly relaxed. In many cases the horizontal alignment comma operator may also be omitted,

```
>> k=[1 4 9:2:17 31];
```

works as well. We shall try to avoid this kind of syntax relaxation although you may encounter it in the help system.

3 Numbers

MATLAB is rather strict with numbers. It adheres to the IEEE[3] convention on representing real numbers and algebraic operations with them. It is obvious that real numbers must be approximated. The approximation, however, should be reliable and as good as possible.

There is a predefined number `eps`. It is the smallest number such that `1` and `1+eps/2` are identical. On my computer `eps=2.2204e-016`. Roughly speaking, the inherent accuracy is 16 digits. This is sufficient for physics where not more than 6 digits are meaningful.

Rounding errors are randomly distributed. With N operations, the error grows proportional to $\sqrt{N}$. Hence, 10^{20} operations on real numbers are allowed until the statistical error exceeds the 6 digits limit. If your computer runs at 1.0 GHz, approximately 10^{11} seconds are required to perform such a large number of operations. This is more than 1000 years. Put otherwise: rounding errors, unless crucial, will not pose a problem.

Rounding errors become crucial if a result is the difference between two large, but almost equal numbers. Try

```
>> x=1e-15;
>> y=((1+x)-1)/x
```

[3]Institute of Electrical and Electronics Engineers

The result is 1.1102, but should be 1.

Assume you want to calculate $f(x) = \sin(x)/x$ for $x \in [0, 2\pi]$. The following code does is:

```
>> x=linspace(0,2*pi,1024);
>> y=sin(x)./x;
```

Typing

```
>> y(1)
```

produces `NaN`, not a number. And this should be so. After all, we have divided zero by zero, and the result is neither 1 nor infinite, it is not defined.

The IEEE standard of floating point representation, which is realized by all Intel processors and supported by MATLAB software, provides for a bit pattern which is not a valid number. Any operation with `NaN` results in `NaN`. `NaN`s tend to spread out. For example,

```
>> z=y/sum(y);
```

results in a vector filled with `NaN`s.

A `NaN` can be detected by the function `isnan`. `isnan(y(1))` returns 1 which stands for true. `isnan(y(2))` results in 0, or false. `isnan(y)` returns an vector the first component of which is 1, the remaining are 0.

Another special number of the IEEE standard is `Inf` which stands for positive infinity. `1/0` results in `Inf`, `-1/0` in `-Inf`, just as `log(0)`. `Inf/Inf` or `Inf-Inf` are ill defined, i. e. `NaN`. However, `Inf+Inf` gives `Inf`.

The pseudo-numbers `Inf` and `NaN` solve a long-standing problem. In the old days, you could either demand program abort if an ill-defined arithmetic operation was performed, or the result was arbitrary. With the IEEE standard and high quality software the program is not halted, but errors are realized and documented.

Let us return to $f(x) = \sin(x)/x$. This function, by declaring $f(0) = 1$, can be made to be continuous everywhere.

```
>> y=sin(x)./x;
>> y(1)=1;
```

is a plausible and obvious solution.

However, you have made use of the fact that it is the first component which is exceptional. Much better is the following solution:

```
>> y=ones(x);
```

```
>> k=find(x~=0);
>> y(k)=sin(x(k))./x(k);
```

`find` returns a vector of indices for which `x` is not equal to zero.

A third possibility[4] is old-style procedural programming:

```
1   for k=1:1024
2     if x(k)==0
3       y(k)=1;
4     else
5       y(k)=sin(x(k))/x(k);
6     end
7   end
```

This is not recommended. Loops in MATLAB are slow, they should be avoided by all means.

A fourth possibility would be

```
>> x=linspace(eps,2*pi,1024);
```

Although it works, this is a dirty trick. The problem is avoided, not solved.

4 Functions

Functions map one ore more real or complex variables into real or complex numbers. Functions are essential for physics, they describe relationships between measurable quantities.

Let us study an example. The spectral intensity of black body radiation is described by Planck's famous formula

$$S(x) = \frac{15}{\pi^4} \frac{x^3}{e^x - 1} \quad , \tag{1}$$

where x is short for $\hbar\omega / k_\mathrm{B} T$.

(1) is good praxis. Computational physics has to deal with numbers, i. e. dimensionless quantities. The spectral density is a probability distribution, a dimensionless quantity. In fact,

$$\int_0^z dx\, S(x) = \Pr\{\hbar\omega < z k_\mathrm{B} T\} \quad . \tag{2}$$

[4] Line numbers instead of the `>>` prompt indicate that we have executed a script file, `test.m` in this case

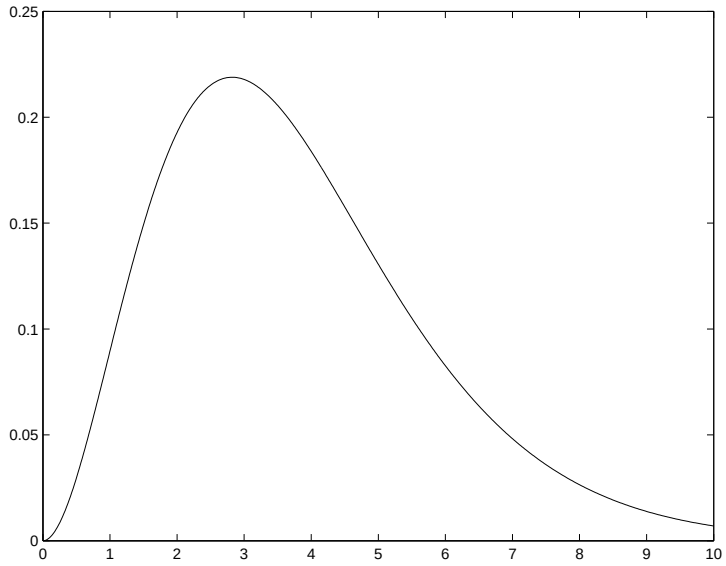

Figure 1: The black body radiation spectral intensity $S(x)$ where $x = \hbar\omega/k_{\mathrm{B}}T$

In MATLAB you describe a function in a separate m-file. *The filename without the* `.m` *extension is the name of the function.* (1) is realized as follows:

```
1   % this file is planck.m
2   % spectral intensity of black body radiation
3   function s=planck(x);
4   s=zeros(size(x));
5   k=find(x~=0);
6   s(k)=15/pi^4*x(k).^3./(exp(x(k))-1);
```

We may now say

```
>> x=linspace(0,10,1024);
>> plot(x,planck(x));
>> print -deps2 planck.eps
```

Fig. 1 shows the result.

The figure has been exported to an Encapsulated Postscript (level 2) file `planck.eps`. It was transformed into `planck.pdf` by the `eps2pdf` program and included into this LaTeX document by saying

```
\FIG{planck.pdf}
{The black body radiation spectral intensity $S(x)$
```

```
where $x=\hbar\omega/\kB T$.}
```

My `\FIG` macro is defined as follows:

```
1   \newcommand{\FIG}[2]{
2   \begin{figure}[!hbt]
3   \begin{center}
4   \begin{minipage}{0.85\textwidth}
5   \centering{\includegraphics[width=95mm]{#1}}
6   \caption{\label{#1}\small{#2}}
7   \end{minipage}
8   \end{center}
9   \end{figure}
10  }
```

You must have imported the `epsfig` package before.

We have explained how to define and how to visualize a function. What else can you do?

Let us numerically check whether (1) is indeed a probability distribution. We approximate $x = \inf$ by $x = 20$, say. The following line performs the integration:

```
>> quad(@planck,0,20)
```

The output 0.99999729919447 (with `format long`) is convincing since the quadrature method `quad` has a predefined tolerance of six digits. And 20 is not yet infinity. `quadl` is better, although slower. The following call

```
>> quadl(@planck,0,40)
```

results in 1.00000000003192.

The function, here `planck`, is referred to by its handle `@planck`.

```
>> quad('planck',0,20)
```

will work as well. Although there are subtle differences, we shall not discuss them here.

`quad` or `quadl` are in-built functions which work on functions. You may inquire by typing

```
>> help funfun
```

`fminbnd` is such a function function. It finds out the position of the minimum within prescribed bounds. However, we want to know about the maximum position of our planck function. Therefore we must define the negative of

the planck probability distribution. For this we do not require a new m-file, because there is the `inline` statement:

```
>> np=inline('-planck(x)');
```

The string is parsed as we would read it.

```
>> xmax=fminbnd(np,0,20)
```

delivers 2.8214.

We now may ask

```
>> elo=quad(@planck,0,xmax);
>> ehi=quad(@planck,xmax,20);
```

for finding the amount of energy below and above the spectral intensity maximum. The two numbers should add up to 1. Do they?

5 Ordinary differential equations

We talk about a system which is in a certain state. If the system has N degrees of freedom, the state is described by a vector $y_1, y_2, \ldots, y_N$ of variables. The state will change in the course of time. In many cases the time development of the state of a system is adequately described by a system of ordinary differential equations,

$$\dot{y}_j = f_j(t, y_1, y_2, \ldots, y_N) \quad . \tag{3}$$

Higher than first derivatives can always be removed. After all, the second derivative is the first derivative of the first derivative, and so forth.

Let us investigate one of the oldest physical problems, the motion of planets in the sun's gravitational field.

Recall that the conservation of angular momentum requires the planet to move in a plane. The planet's location $x_1(t), x_2(t)$ obey the following differential equation:

$$m\ddot{x}_j(t) = -\frac{GmM_\odot\, x_j}{(x_1^2 + x_2^2)^{3/2}} \quad . \tag{4}$$

G is the universal gravitational constant, m the planet's and $M_\odot$ the sun's mass. m drops out. By measuring x in units of the astronomical unit[5] a

[5] the mean distance between earth and sun, 149.6×10^{11} m

and t in units of $\tau = \sqrt{a^3/GM_\odot}$ we arrive at[6]

$$\ddot{x}_j(t) = -\frac{x_j}{(x_1^2 + x_2^2)^{3/2}} \quad . \tag{5}$$

Let us introduce $(y_1, y_2, y_3, y_4) = (x_1, \dot{x}_1, x_2, \dot{x}_2)$ such that

$$\begin{aligned} \dot{y}_1 &= y_2 && (6)\\ \dot{y}_2 &= -y_1/r^3 && (7)\\ \dot{y}_3 &= y_4 && (8)\\ \dot{y}_4 &= -y_3/r^3 && (9) \end{aligned}$$

results, where $r = \sqrt{y_1^2 + y_3^2}$.
We describe this set of four ordinary differential equations by the following derivative field:

```
1  % this file is kepler.m
2  function d=kepler(t,y);
3  r=sqrt(y(1).^2+y(3).^2);
4  d=[y(2);-y(1)./r^3;y(4);-y(3)./r^3];
```

At the command window we say

```
>> [t,y]=ode45(@kepler,[0:0.1:50],[1;0;0;0.8]);
```

A Runge-Kutta integration procedure `ode45` is invoked. It needs a derivative field, a time span, and a start vector. And this is the result of saying

```
>> plot(y(:,1),y(:,3))
>> print -deps2 kepler1.eps
```

Although the trajectories are ellipses, they shrink more and more. The reason is not physics, but numerics. We substantiate this remark by plotting the energy in Fig. 3

```
>> Ekin=0.5*(y(:,2).*y(:,2)+y(:,4).*y(:,4));
>> Epot=-1./sqrt(y(:,1).*y(:,1)+y(:,3).*y(:,3));
>> plot(t,Ekin+Epot);
```

We try better by setting a lower relative tolerance (the default being 0.001):

[6] $2\pi\tau$ is one year

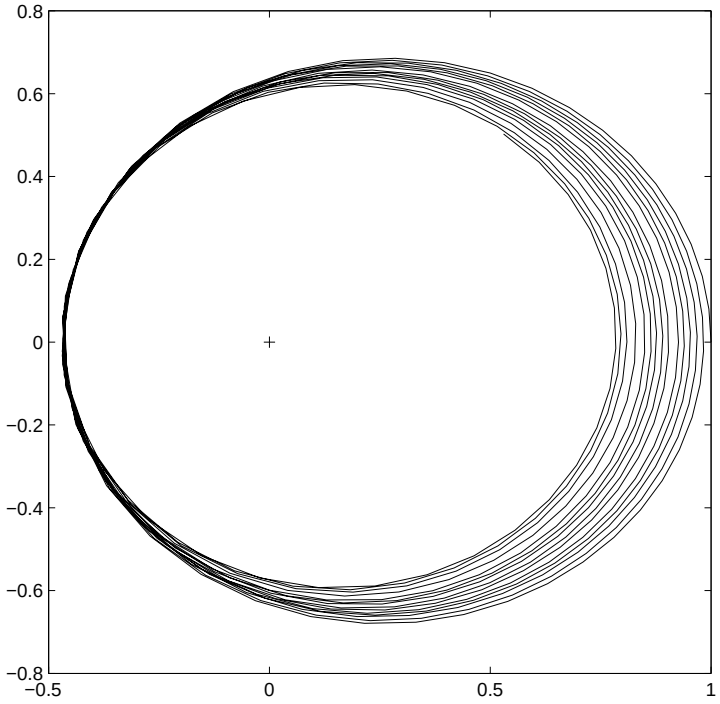

Figure 2: Planetary motion without explicite accuracy control

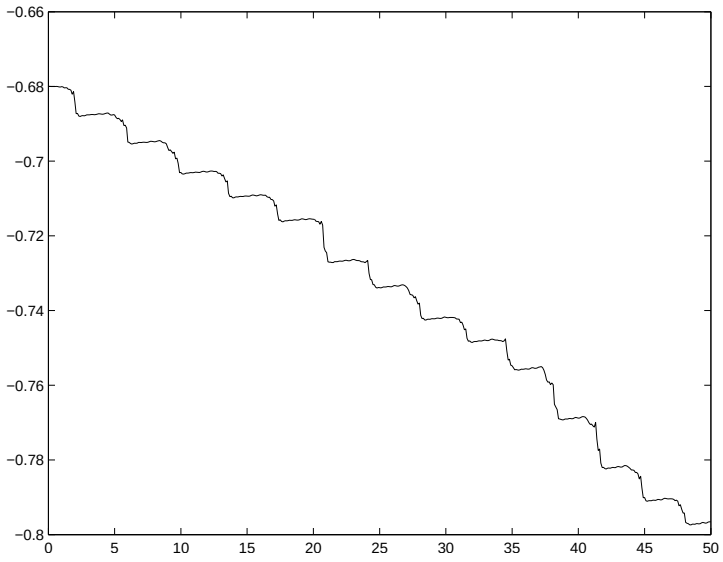

Figure 3: Total energy vs. time without accuracy control

```
>> tol=odeset('RelTol',1e-8);
>> [t,y]=ode45(@kepler,[0:0.1:50],[1;0;0;0.8],tol);
```

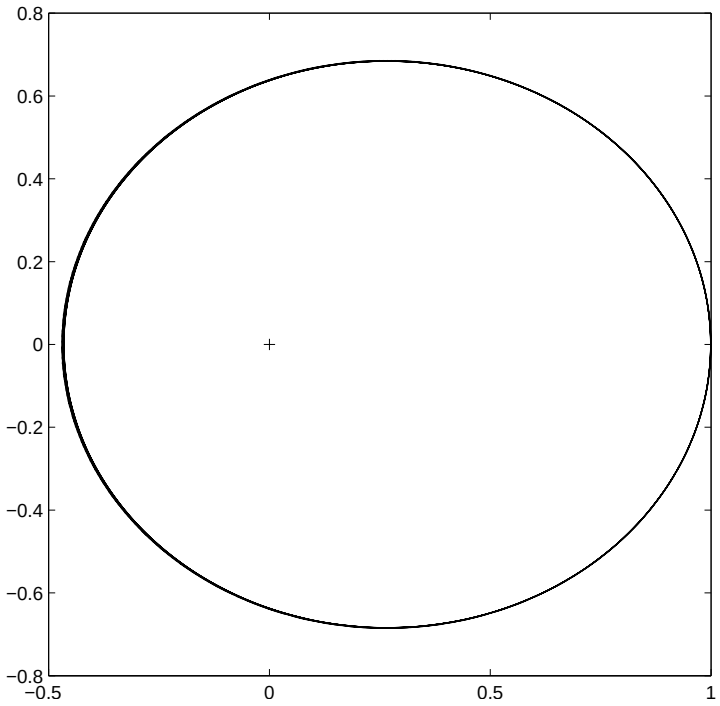

Figure 4: Planetary motion with accuracy control

Fig. 4 shows what we expect: closed ellipses.

There are many more options for the ordinary differential equation solver, and there are more such solvers. You should ask for details at the help desktop.

6 Fast Fourier transform

There are three kinds of Fourier transforms: finite discrete, infinite discrete, and continuous. The first is defined by

$$G_j = \sum_{k=0}^{N-1} e^{-2\pi ijk/N} g_k = \sum_{k=0}^{N-1} \Omega_{jk} g_k \tag{10}$$

for $j = 0, 1, \ldots, N-1$. The Matrix Ω is given

$$\Omega_{jk} = \omega^{-j \cdot k} \text{ where } \omega = e^{2\pi i/N} . \tag{11}$$

ω is the Nth root of 1.
Because $\Omega/\sqrt{N}$ is unitary,

$$\sum_{k=0}^{N-1} \Omega_{jk}(\Omega^{\dagger})_{kl} = N\delta_{j,l} \tag{12}$$

we obtain for the inverse transform the following expression:

$$g_k = \frac{1}{N}\sum_{j=0}^{N-1} e^{+2\pi ijk/N} G_j \ . \tag{13}$$

Only the finite discrete Fourier transform is of interest in computational physics since a continuum has to be approximated by an interval, and an interval by a finite set of representative points.
To be specific, we think about a time interval, $t_k = k\tau$ for $k = 0, 1, \ldots, N-1$. $f_j = j/N\tau$ are frequencies, and we may rewrite (10) and (11) into

$$G_j = G(f_j) = \sum_{k=0}^{N-1} e^{-2\pi i f_j t_k} g_k \tag{14}$$

and

$$g_k = g(t_k) = \frac{1}{N}\sum_{j=0}^{N-1} e^{2\pi i f_j t_k} G_j \ . \tag{15}$$

Let us simulate a very noisy cosine signal. We set

```
1   % this file is noisy_cos.m
2   fbar=50;
3   tau=0.001;
4   N=1024;
5   t=tau*[0:N-1];
6   R=2.0;
7   g=cos(2*pi*fbar*t)+R*randn(size(t));
8   plot(t,g,'.');
9   axis([min(t),max(t),-4,4]);
10  print -deps2 ncos1.eps;
```

Would you recognize the cosine signal in Fig. 5?
We continue by implying the fast Fourier transform `fft`:

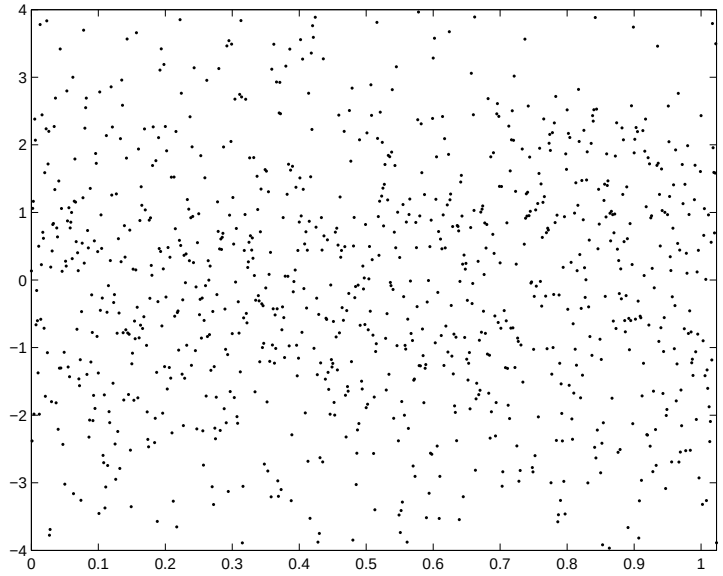

Figure 5: Noisy cosine (50 Hz) vs. time sampled at millisecond steps

```
11  G=fft(g);
12  f=[0:N/2-1]/N/tau;
13  S=abs(G(1:N/2)).^2;
14  plot(f,S);
15  print -deps2 ncos2.eps;
```

In Fig. 6 we have plotted the spectral power $S = |G(f)|^2$ for positive frequencies[7]. The prominent peak at $f = 50$ Hz is evident. The remaining spectral power is more or less constant which is indicative of white noise.

How can one extract the signal so convincingly from a lot of noise? Not by looking at the sampled data. They appear to be random. However, by performing a Fourier analysis, the different harmonic contributions are excited with their proper phases, so that they add up.

If you know that the signal is spoilt by white noise, you may remove it to a large extent. You might say

```
>> H=G.*(abs(G)>150);
>> h=ifft(H);
```

`ifft` denotes the inverse fast Fourier transform as described by (13).

You can do a lot more with the fast Fourier transform. Here are some examples:

[7]The spectral power is an even function of f

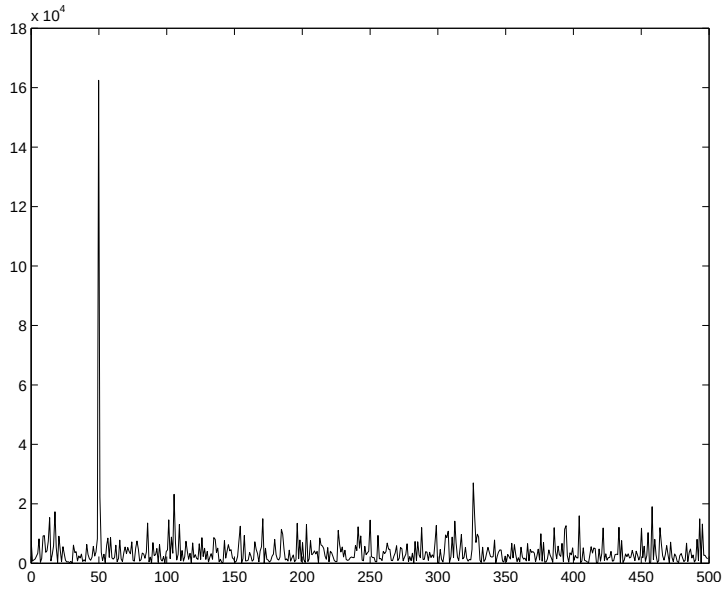

Figure 6: Spectral power of noisy cosine vs. frequency (Hz)

- Differentiation: Fourier transform the signal and multiply by $2\pi if$, and back Fourier transform. Thereby a much better precision can be achieved than by difference quotients.
- Deconvolution: Often the output $b(t) = \int ds\, r(s)a(t-s)$ is the convolution of a signal a and a transmission function r. If the transmission function is known, then, from $B(f) = R(f)A(f)$, the signal can be reconstructed. Just divide the Fourier transformed output by the Fourier transformed transmission function, and back Fourier transform.
- Differential equations with constant coefficients: Differentiation operators become multiplication operators after Fourier transformation, and differential equations become algebraic equations.
- Data filtering and smoothing: remove the high frequency components.

The fast Fourier transform is an algorithm which takes into account that an operation on $2N$ data points are two operations on N data points. This reduces complexity from N^2 to $N\log(N)$ which makes all the difference.

7 Fitting data

Measured data are to be compared with a theoretically justified model. Discrepancies may arise for two different reasons: the data are not accurate, or

the model is inappropriate. Inaccurate data may arise because of systematic or statistical errors.

Let us first formulate the standard task.

You have a set of data $(y_1, x_1), (y_2, x_2), \ldots (y_N, x_N)$ and a model $y = f_p(x)$. The p denote a set of parameters which are to be determined. For this we define the variance

$$v(p) = \frac{1}{N-1} \sum_{i=1}^{N} (f_p(x_i) - y_i)^2 \quad . \tag{16}$$

The optimal fit is defined by minimal variance,

$$v(\bar{p}) = \min_p v(p) \quad . \tag{17}$$

If the model is a linear function, we speak of linear regression. Likewise, if it a polynomial of degree 2, the data are fitted by quadratic regression, and so forth.

With MATLAB this is very easy. If you say `p=polyfit(x,y,d);` a polynomial `p` of degree `d` is returned which best fits the data vectors `x` and `y`. As mentioned earlier, a polynomial is represented by the vector of its coefficients.

Here is an example:

```
1   % this file is fit_regr.m
2   x=linspace(0,2,256);
3   y=1-x+0.5*x.^2+0.15*randn(size(x));
4   plot(x,y,'.k');
5   hold on;
6   p=polyfit(x,y,2);
7   yy=polyval(p,x);
8   plot(x,yy,'-k','LineWidth',2);
9   hold off;
10  print -deps2 fitr.eps;
```

We have plotted the convincing result in Fig. 7.

If the model is not a polynomial, one has to minimize numerically the variance or any other measure of the misfit.

Let us assume a Gaussian peak on top of background. The model is

$$y = y_0 + s\, e^{-a(x-x_0)^2} \quad . \tag{18}$$

We have four parameters to fit, namely $p = (y_0, s, a, x_0)$. Let us work out an example.

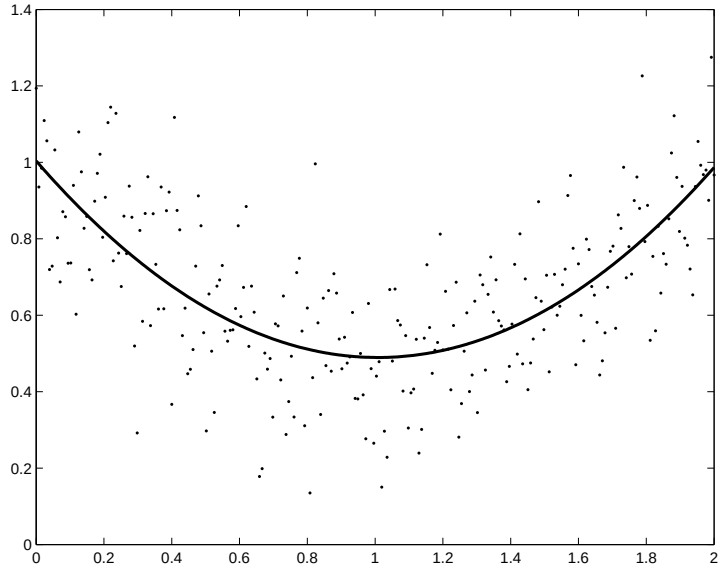

Figure 7: Noisy quadratic relationship between x (abscissa) and y (ordinate) and reconstruction by quadratic regression

```
1  % this file is fit_peak
2  peak=@(p,x) p(1)+p(2)*exp(-p(3)*(x-p(4)).^2);
3  misfit=@(p,x,y) norm(y-peak(p,x));
```

The first command defines the model, a Gaussian peak. It is the MATLAB equivalent of (18). The next line describes the misfit of a data set `x,y` to the model, for a certain parameter set `p`. We will next simulated data:

```
4  tp=[3;1;4;2.5];
5  x=linspace(0,5,1024);
6  ty=peak(tp,x);
7  ny=ty+0.5*randn(size(x));
```

The true parameter set `tp` gives rise to `ty=peak(tp,x)`. We add noise to it and obtain the noisy data values `ny`. From these noisy data we reconstruct the best fitting parameters `fp` by saying

```
8  fp=fminsearch(misfit,tp,[],x,ny);
```

The first argument to `fminsearch` is the misfit, or cost function. The second argument is a parameter set from which to begin the search. Then comes an options structure for which we specify nothing, i. e. `[]`. In this case the default values for `fminsearch` are used.

Normally the cost function has just one argument, namely the vector of variables over which to minimize. If the cost function requires more arguments (which remain constant when searching for a minimum), these are to be specified in the remainder of the argument list of `fminsearch`. In our case it is the data set `x,ny` to be fitted.

Let us plot the result:

```
9   fy=peak(fp,x);
10  plot(x,ty,'-k',x,ny,'.b',x,fy,'-r','LineWidth',2);
```

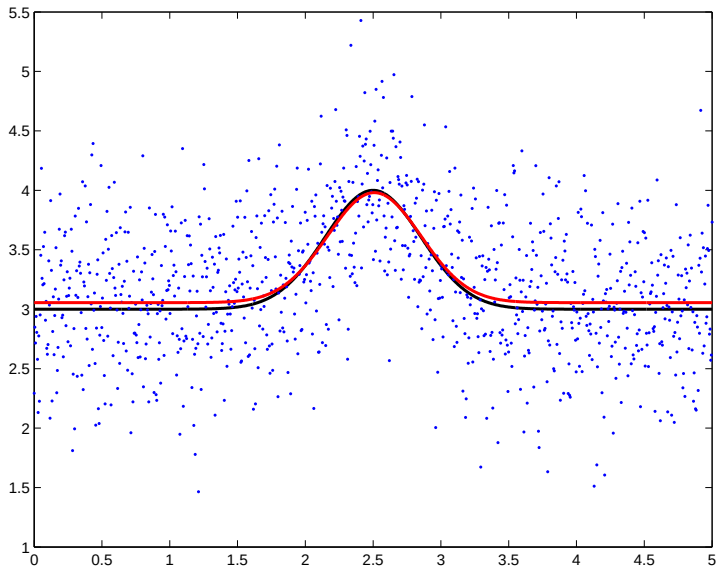

Figure 8: A Gaussian peak on top of background: signal (black), signal plus noise (blue dots) and reconstructed signal (red)

However, what will be the result if the search for the best fit starts from another parameter set which is not so close?

Let us try `sp=[1;1;1;1]` as the starting parameter set. This will work. However, with `sp=[0;0;0;0]` the minimization procedure does not converge to a sensible fit.

We might try to fiddle with the optimization options:

```
11  opt=optimset('TolFun',1e-8,'TolX',1e-8,...
12      'MaxFunEvals',20000,'MaxIter',10000);
13  fp=fminsearch(misfit,[0;0;0;0],opt,x,ny);
```

This helps occasionally[8], but more often not. Obviously the minimization

[8] bear in mind that we solve a different problem each time because of randoms in the

procedure runs into this or another shallow side minimum and misses the absolute minimum.

You might program a coarse pre-search such as this:

```
% this file is pre_search.m
mm=Inf;
for j=1:10000
    p=[6*rand;2*rand;8*rand;5*rand];
    m=misfit(p,x,ny);
    if (m<mm)
        mm=m;
        pp=p;
    end;
end;
```

It should deliver a good starting parameter set `pp`. The cuboid $0 \leq p_1 \leq 6$, $0 \leq p_2 \leq 2$ etc. has been specified after inspecting the data cloud.

8 Simulated Annealing

Minimization problems are among the most frequently encountered tasks of computational physics. There is a set of parameters, and each parameter set is weighted by a cost function. The problem to be solved is: find the parameter set for which the costs are minimal.

We did already discuss such an optimization problem in the context of model fitting. The Nelder-Mead simplex method (`fminsearch`) is never the best, but always a good choice to tackle the problem. Remember: you have to specify a starting point when searching for the minimum. In general, the local minimum closest to the starting point is found.

If there are very many parameters, there are also very many local minima, but there is only one global minimum. Therefore, a coarse search for a sensible starting point is the first step in order to find a sensible starting point. In many cases this first step is visual inspection of data, or intuition, or the result of previous minimization efforts.

There are however problems where there are so many local minima that a direct search for the global minimum is appropriate.

Here we discuss a computational classic: the traveling salesman problem.

There are `NC` cities to be visited, their Cartesian coordinates are stored in vectors `XC` and `YC`, respectively. The itinerary is described by a a permutation of indices from `1` to `NC`, a vector `it`. The cost function is the length of a round trip, as calculated by

data set

```
1   % this file is ts_length.m
2   function len=ts_length(it);
3   global NC XC YC
4   itt=it([2:NC,1]);
5   len=sum(sqrt((XC(itt)-XC(it)).^2+(YC(itt)-YC(it)).^2));
```

The `global` statement says that all or some of the variables may be visible to subprograms or functions if they want to see them.

We silently assume that the effort to travel between two cities is proportional to their distance. This can be easily modified by introducing a table which describes the effort to travel from one to another city.

The number of possible itineraries is finite, but prohibitively large. If there are only 20 cities, we have to check almost 10^{17} possibilities. If one check lasts only 1 μs, this requires 10^{11} s, almost 3000 years.

Here we describe the simulated annealing algorithm[9]. A probe is heated and then slowly cooled down. Thereby all sorts of defects may be mended. Temperature allows for random fluctuations, and cooling will lead to the state of lowest energy. Even if the system is close to a local energy minimum, a fluctuation may send it to an even better minimum.

If a new configurations has a lower energy (cost), it is accepted straight away. If its energy is higher, it shall be accepted with probability $\exp(-\Delta E/T)$ where T is the temperature and ΔE the energy increase.

So we set up a sensible temperature, allow for random variations of the itinerary, and let the system cool down.

Here is a not-yet optimized algorithm.

```
1   % this file is ts_problem.m
2   global NC XC YC
3   NC=20;
4   XC=rand(1,NC);
5   YC=rand(1,NC);
6   T=ts_hot;
7   suc=1;
8   it=[1:NC];
9   while suc>0
10    [it,suc]=ts_anneal(T,it);
11    T=0.9*T;
12  end
```

We simulate the locations of 20 cities, set up an initial temperature T, choose city 1 to city 2 to ... as an initial itinerary, and let it cool down in steps of

[9] Adapted from W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, Numerical Recipes, Cambridge University Press 1986, ISBN 0521308119

10 % temperature reduction. For each temperature, the system is annealed. The relevant function returns the new best itinerary and the number of successful trials to find a shorter path. The program stops if there are no more improvements.

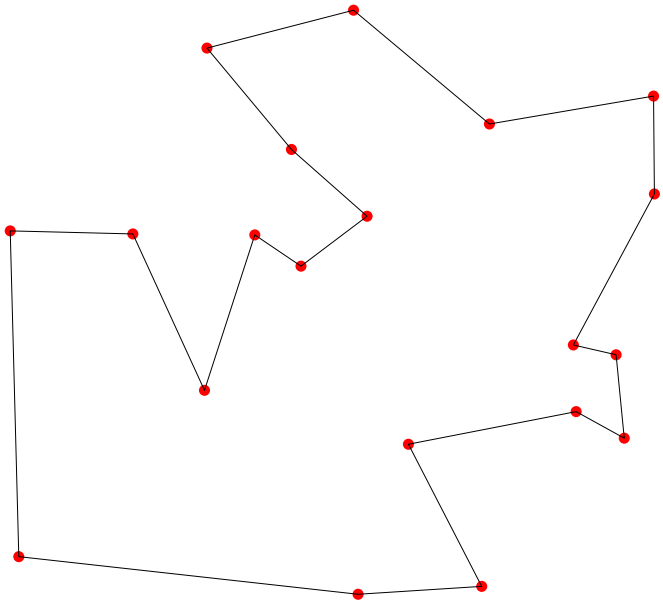

Figure 9: The shortest itinerary found by the simulated annealing algorithm

Fig. 9 shows the solution. It was created by calling `ts_problem` (see above).

The `anneal` function lets the system perform random fluctuations. If a modified itinerary is shorter, it is accepted right away. If it is longer, but not too much, it will be accepted as well. How much is too much is ruled by the temperature. The lower the temperature, the likelier a worsening will be rejected.

Here is the code:

```
1   % this file is ts_anneal.m
2   function [ii,suc]=ts_anneal(T,it);
3   global NC
4   max_rep=100*NC;
5   max_suc=10*NC;
6   suc=0;
7   ii=it;
8   min_len=ts_length(it);
9   for rep=1:max_rep
```

```
10      im=ts_modify(it);
11      len=ts_length(im);
12      if rand<exp(-(len-min_len)/T)
13        ii=im;
14      end;
15      if len<min_len
16        min_len=len;
17        suc=suc+1;
18      end
19      if suc==max_suc
20        break
21      end
22    end
```

We try at most `max_rep` times. And if the number of successes is too large, we stop as well and try a lower temperature. The decisive statement is

```
>> if rand<exp(-(len-min_len)/T)
```

expressing that we accept a modification if the new length is smaller than the up-to-now lowest length, or if the new length is not too much larger. This is not yet, but might become an improvement because the modification may be close to a better local minimum.

It remains to be explained how itineraries are modified at random. We have programmed two alternatives. We cut out a sub-itinerary and revert it, with 50 % probability, or we transport it by a random amount. This is formulated here:

```
1   % this file is ts_modify.m
2   function im=ts_modify(it);
3   global NC
4   n=sort(ceil(NC*rand(1,2)));
5   ib=it(1:n(1)-1);
6   is=it(n(1):n(2));
7   ia=it(n(2)+1:NC);
8   if rand<0.5
9     im=[ib,fliplr(is),ia];
10  else
11    ir=[ib,ia];
12    len=length(ir);
13    n=ceil(len*rand);
14    im=[ir(1:n),is,ir(n+1:len)];
15  end;
```

The itinerary `it` is split into a randomly chosen sub-itinerary `is`, its leader

`ib` and trailer `ia`. `fliplr` flips matrices in left-right direction. Note that this simple code does not guarantee that the original and the modified itinerary differ.

In order to be complete, we also show how the initial temperature was chosen:

```
1   % this file is ts_hot.m
2   function T=ts_hot();
3   global NC
4   lmax=0;
5   for k=1:20
6     len=ts_length(randperm(NC));
7     if len>lmax
8       lmax=len;
9     end;
10  end;
11  T=2*lmax;
```

Calling `randperm(N)` returns a random permutation of `[1:N]`.

9 Finite difference method

The finite difference method of treating differential equations is simple. The limit in

$$f'(x) = \lim_{h\to 0} \frac{f(x+h/2) - f(x-h/2)}{h} \tag{19}$$

is approximated by a finite difference quotient. Likewise, the second derivative is to be approximated by

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \quad . \tag{20}$$

The Laplacian in two dimensions is approximated by

$$\frac{f(x+h,y) + f(x,y+h) + f(x-h,y) + f(x,y-h) - 4f(x,y)}{h^2} \quad . \tag{21}$$

Generalizations to different step widths h_x, h_y and for more than two dimensions are obvious.

Our first example is a simple ordinary differential equation:

$$f'' = -f \;\text{ with }\; f(0) = 0 \;\text{ and }\; f(\pi/2) = 1 \quad , \tag{22}$$

the solution of which is $f(x) = \sin(x)$.

The normal ordinary differential equation solvers require initial conditions, such as $f(0) = 0$ and $f'(0) = 1$. We, however, are faced with boundary conditions.

The following program realizes the finite difference method.

```
1   % this file is fdm_sin.m
2   N=16;
3   h=pi/2/N;
4   next=(1/h^2)*ones(1,N-2);
5   main=(-2/h^2+1)*ones(1,N-1);
6   DE=diag(next,-1)+diag(main,0)+diag(next,1);
7   RS=zeros(1,N-1)';
8   RS(N-1)=-1/h^2;
9   sol=DE\RS;
10  x=[h:h:(N-1)*h];
11  xx=linspace(0,pi/2,256);
12  plot(x,sol,'r.',xx,sin(xx),'b-',[0,pi/2],[0,1],'k.','LineWidth',2,'MarkerSize',20
13  axis tight;
14  print -depsc fdm_sin.eps;
```

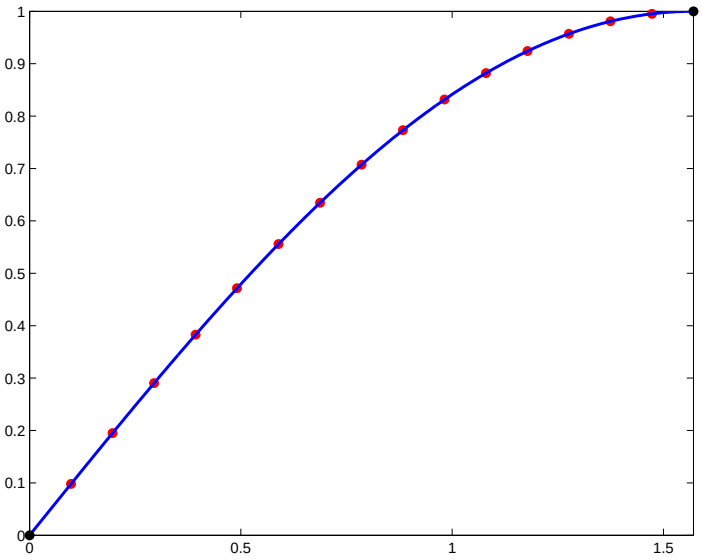

Figure 10: Analytic solution (full line) and finite difference approximation (red) of $f'' + f = 0$ with $f(0) = 0$ and $f(\pi/2) = 1$ (black)

We have plotted the result in Fig. 10. The maximum deviation is 2×10^{-4} although only 15 representative points have been used. Well, better say

17 because the entries `RS(1)=0` and `RS(N-1)=-1/h^2` reflect the boundary conditions. The second derivatives at the smallest and largest interior point require information from outside.

Note the `diag` function which allows to set and to extract diagonals. Also note that `x=M\y` is short for solving the system of linear equations `y=M*x`. `axis tight` does what it says: the axis frame is fitted as tightly as possible to the data.

In our second example we want to solve a two-dimensional boundary problem. Let us reproduce the MATLAB logo which is a visualization of the lowest order eigenmode of the following equation:

$$-\Delta u = \Lambda u \quad . \tag{23}$$

$u = u(x, y)$ is defined on a set $\Omega \subset \mathbb{R}^2$ with appropriate conditions on the boundary $\partial\Omega$. It is a challenge to solve this partial differential eigenvalue problem on the L-shaped domain

$$\Omega = [-1, 1] \times [-1, 1] - [-1, 0] \times [-1, 0] \tag{24}$$

with $u = 0$ on $\partial\Omega$.

We shall not describe the most efficient solution but a program which is easy to generalize.

We begin by describing the domain.

```
1   % this file is ml_logo.m
2   N=32;
3   h=2/(N-1);
4   x=linspace(-1,1,N);
5   y=linspace(-1,1,N);
6   [X,Y]=meshgrid(x,y);
7   Domain=(abs(X)<1)&(abs(Y)<1)&((X>0)|(Y>0));
```

`X` and `Y` are $N \times N$ matrices such that `X(j,k)=x(k)` and `Y(j,k)=y(j)`. `Domain` is likewise an $N \times N$ matrix with 1 for an interior point and 0 for a boundary or exterior point.

A subprogram `laplace` will calculate the sparse matrix `L` approximating the Laplacian Δ by (21). It also returns the mapping of the running index `a` for variables to pairs j, k which index the x, y coordinates. L is defined by $(\Delta u)_a = \sum_b L_{ab} u_b$. We have $L_{aa} = -4/h^2$ and $L_{ab} = 1/h^2$ whenever a and b index neighboring variables.

The rest is easy. We call the sparse matrix eigenvalue solver `eigs` which returns the lowest order eigensolution `u` and the corresponding eigenvalue `d`:

```
8   [L,J,K]=laplace(Domain,h);
9   [u,d]=eigs(-L,1,'sm');
10  s=sign(sum(u));
11  field=zeros(size(Domain));
12  for a=1:size(u)
13    field(J(a),K(a))=s*u(a);
14  end;
15  mesh(field);
16  axis off;
17  print -depsc ml_logo.eps;
```

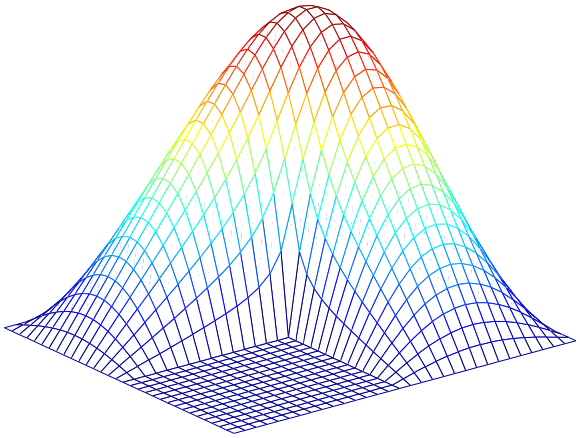

Figure 11: The lowest order eigenmode of $-\Delta u = \Lambda u$ on an L-shaped domain has become their logo

The `laplace` function which calculates the Laplacian as a sparse matrix and returns the indexing scheme is here:

```
1   % this file is laplace.m
2   function [L,J,K]=laplace(D,h);
3   [Nx,Ny]=size(D);
4   jj=zeros(Nx*Ny,1);
5   kk=zeros(Nx*Ny,1);
6   aa=zeros(Nx,Ny);
7   Nv=0;
8   for j=1:Nx
9     for k=1:Ny
```

```
10        if D(j,k)==1
11          Nv=Nv+1;
12          jj(Nv)=j;
13          kk(Nv)=k;
14          aa(j,k)=Nv;
15        end
16      end
17    end
18    L=sparse(Nv,Nv);
19    for a=1:Nv
20      j=jj(a);
21      k=kk(a);
22      L(a,a)=-4/h^2;
23      if D(j+1,k)==1
24        L(a,aa(j+1,k))=1/h^2;
25      end
26      if D(j-1,k)==1
27        L(a,aa(j-1,k))=1/h^2;
28      end
29      if D(j,k+1)==1
30        L(a,aa(j,k+1))=1/h^2;
31      end
32      if D(j,k-1)==1
33        L(a,aa(j,k-1))=1/h^2;
34      end
35    end
36    J=jj(1:Nv);
37    K=kk(1:Nv);
```

In our case, we have 675 independent variables, hence `L` should require 3.6 MB of storage. However, most of its entries are zeros. Therefore it is sufficient to store the locations and values of the non-vanishing entries only, 42 kB in this case. MATLAB provides this sparse matrix technology which is essential for partial differential equations and for many other branches of computational physics.

10 Propagation

In the previous section we have studied a special class of partial differential equations. The solution was fixed by values on a boundary. Here we address another class of problems. A field is prescribed at time $t = 0$. How will it propagate in the course of time?

Let us study a simple, but typical[10] partial differential equation of this type, namely

$$\dot{u} = \Delta u \ , \tag{25}$$

for simplicity in one spatial dimension, $u = u(t, x)$. The dot denotes partial differentiation with respect to the time argument t, the Laplacian is the second derivative with respect to the space argument x. The spatial region is an interval, $0 \leq x \leq 1$, say. Time begins with $t = 0$. We look for a solution of (25) subject to the following conditions:

$$u(0, x) = u_0(x) \ \text{ as well as } \ u(t, 0) = b_0(t) \ \text{ and } \ u(t, 1) = b_1(t) \ . \tag{26}$$

$u_0 = u_0(x)$ is an initial condition, $b_k = b_k(t)$ for $k = 0, 1$ are boundary conditions. We assume that these restrictions are compatible, i. e. $u_0(0) = b_0(0)$ and $u_0(1) = b_1(0)$.

We resort to the finite difference method and write

$$u_{j,k} = u(j\tau, kh) \tag{27}$$

for integers j and k and time and space steps τ and h, respectively.

The most natural approach is to approximate (25) by

$$u_{j+1,k} = u_{j,k} + \tau \, \{ \frac{u_{j,k+1} - 2u_{j,k} + u_{j,k-1}}{h^2} \} \ . \tag{28}$$

If the variables are labeled by $k = 1, 2, \ldots, N$, then $u_{j,0}$ in (28) has to be replaced by $b_{0,j} = b_0(j\tau)$ and $u_{j,N}$ by $b_{1,j} = b_1(j\tau)$. The differencing scheme is fully explicite since it allows to calculate new field values in terms of previously known values. One may write

$$u_{j+1} = (I + \tau L) u_j \ . \tag{29}$$

It turns out that (28) is stable[11] only if $\tau < h^2$. Hence, with $h = 0.01$, we must proceed in time steps of $\tau = 10^{-4}$ or smaller. This is not acceptable.

Another, fully implicite differencing scheme is

$$u_{j,k} = u_{j+1,k} - \tau \, \{ \frac{u_{j+1,k+1} - 2u_{j+1,k} + u_{j+1,k-1}}{h^2} \} \ , \tag{30}$$

or

$$u_j = (I - \tau L) \, u_{j+1} \tag{31}$$

[10] heat transport, diffusion, etc.

[11] see the section on *Diffusive Initial Value Problems* in *Numerical Recipes*, loc. cit.

which is solved by

$$u_{j+1} = (I - \tau L)^{-1} u_j \quad . \tag{32}$$

This one is stable for all time steps τ, but we have to solve a system of linear equations for every propagation step. Both differencing schemes are biased: they approximate, by $\dfrac{f(t+\tau) - f(t)}{\tau}$, the time derivative at t or at $t+\tau$, respectively, while it should be at $t + \tau/2$.

We therefore write

$$u_{j+1/2} = (I + \frac{\tau L}{2})\, u_j = (I - \frac{\tau L}{2})\, u_{j+1} \tag{33}$$

which amounts to

$$u_{j+1} = (I - \frac{\tau L}{2})^{-1}(I + \frac{\tau L}{2})\, u_j \quad . \tag{34}$$

This so called *Crank-Nicholson scheme* is stable for all time steps τ and one order more accurate than (29) or (31).

The following MATLAB function propagates the field u at time t by one time step τ according to the Crank-Nicholson scheme. The spacing h and the boundary values $b_k(t + \tau/2)$ (a two component vector) have to be specified as well[12]. Fields are to be represented as column vectors.

```
1   % this file is cn_step.m
2   function v=cn_step(u,tau,h,b);
3   N=length(u);
4   z=0.5*tau/h^2;
5   L=laplacian(N);
6   vv=u+z*L*u;
7   vv(1)=vv(1)+2*z*b(1);
8   vv(N)=vv(N)+2*z*b(2);
9   M=eye(N)-z*L;
10  v=M\vv;
11
12  function L=laplacian(N);
13  next=ones(1,N-1);
14  main=ones(1,N);
15  L=diag(next,-1)-2*diag(main,0)+diag(next,1);
```

Let us study a situation where $u_0(x) = \sin \pi x + \sin 2\pi x$, $b_0(t) = 0$ and $b_1(t) = 0$.

[12] more precisely: the mean of the boundary values before and after the time step

```
% this file is cn_example.m
Nx=500;
x=linspace(0,1,Nx)';
h=x(2)-x(1);
tau=0.0025;
u0=sin(pi*x)+sin(2*pi*x);
u=u0(2:Nx-1);
b=[0,0];
field=u;
for j=1:100
    u=cn_step(u,tau,h,b);
    field=[field,u];
end;
contour(field,32);
print -depsc cn_example.eps;
```

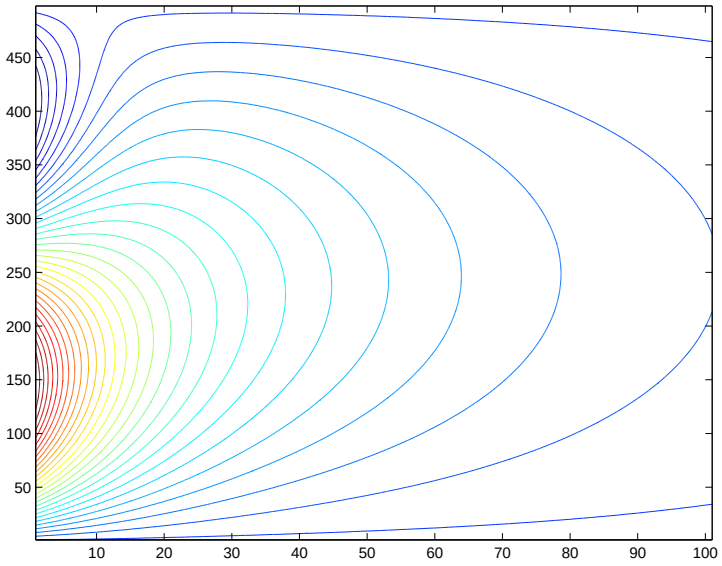

Figure 12: Contour plot of $u = u(t, x)$. Cranck-Nicolson scheme, 100 time propagation steps, t running from left to right. Initially, $u(0, x) = \sin(\pi x) + \sin(2\pi x)$

By the way, Jean Baptiste Joseph Fourier, who lived from 1768 to 1830, has invented the method of decomposing functions into harmonic contributions while trying to solve the heat equation (25). In today's notation:

$$u(t, x) = \sum_{n=1}^{\infty} c_n(t) \sin n\pi x \tag{35}$$

is solved by

$$c_n(t) = c_n(0)\ e^{-(n\pi)^2\, t} \quad . \tag{36}$$

Thus, the `sin(2*pi*x)` contribution dies much more rapidly than the `sin(pi*x)` term as can be read off Fig. 12.

A Figures Makefile

The figures (as `.pdf`-files) were produced with the aid of `.m`-files printed in this document and by the following function. Note that `'!...'` invokes the system command shell.

```
1   % this file is npfigs.m
2   function npfigs();
3
4   % make planck.pdf
5   x=linspace(0,10,512);
6   s=planck(x);
7   plot(x,s,'-k');
8   print -deps2 planck.eps;
9   eval('!epstopdf planck.eps');
10  clear all;
11
12  % make kepler1, kepler2, kepler3
13  [t,y]=ode45(@kepler,[0:0.1:50],[1;0;0;0.8]);
14  plot(y(:,1),y(:,3),0,0,'+k');
15  axis square;
16  print -deps2 kepler1.eps;
17  eval('!epstopdf kepler1.eps');
18  Ekin=0.5*(y(:,2).*y(:,2)+y(:,4).*y(:,4));
19  Epot=-1./sqrt(y(:,1).*y(:,1)+y(:,3).*y(:,3));
20  plot(t,Ekin+Epot);
21  print -deps2 kepler2.eps;
22  eval('!epstopdf kepler2.eps');
23  tol=odeset('RelTol',1e-8);
24  [t,y]=ode45(@kepler,[0:0.1:50],[1;0;0;0.8],tol);
25  plot(y(:,1),y(:,3),0,0,'+k');
26  axis square;
27  print -deps2 kepler3.eps;
28  eval('!epstopdf kepler3.eps');
29  clear all;
30
31  % make ncos1, ncos2
32  rand('state',0);
33  noisy_cos;
34  eval('!epstopdf ncos1.eps');
35  eval('!epstopdf ncos2.eps');
36  clear all;
37
38  % make fitr, fitp
```

```
39  rand('state',0);
40  fit_regr;
41  eval('!epstopdf fitr.eps');
42  clear all;
43  fit_peak;
44  print -depsc fitp.eps
45  eval('!epstopdf fitp.eps');
46  clear all;
47
48  % make ts_problem
49  rand('state',1);
50  ts_problem;
51  plot(XC(it),YC(it),'r.',[XC(it),XC(it(1))],[YC(it),YC(it(1))],...
52      '-k','MarkerSize',20);
53  axis equal;
54  axis off;
55  print -depsc ts_problem.eps;
56  eval('!epstopdf ts_problem.eps');
57  clear all;
58
59  % make fdm_sin
60  fdm_sin;
61  eval('!epstopdf fdm_sin.eps');
62  clear all
63
64  % make ml_logo
65  ml_logo;
66  eval('!epstopdf ml_logo.eps');
67  clear all;
68
69  % make cn_example
70  cn_example;
71  eval('!epstopdf cn_example.eps');
72  clear all;
```

List of Figures