Approximation of functions by exponential sums based on the Newton-type optimisation

by

Wolfgang Hackbusch, Boris N. Khoromskij, and Andreas Klaus

Technical Report no.: 3 2005

Approximation of Functions by Exponential Sums Based on the Newton-type Optimisation

Wolfgang Hackbusch, Boris N. Khoromskij, and Andreas Klaus
Max-Planck-Institute for Mathematics in the Sciences,
Inselstr. 22-26, D-04103 Leipzig, Germany.
{wh, bokh, klaus}@mis.mpg.de

April 26, 2005

Abstract

In this paper we describe and document an algorithm for solving the nonlinear optimisation problem by an adaptive Newton-type method arising from approximation of functions by exponential sums. The algorithm can be applied to efficient representation of multi-dimensional non-local operators and related matrices.

1 Introduction

In a wide range of applications one requires the efficient numerical representation of multi-dimensional integrals and non-local operators/matrices (cf. [5, 2, 3, 4]). In some cases this problem can be reduced to the separable approximation of multi-variate functions in $\mathbb{R}^d$ (cf. [4, 1]). For functions depending on the Euclidean distance in $\mathbb{R}^d$ or on the sum of spatial coordinates, the efficient numerical schemes are based on the quadrature methods or on an approximation by exponential sums.

For example, one may consider separable representations of the multi-variate functions

$$f_1(x_1, ..., x_d) := \frac{1}{x_1 + ... + x_d}, \quad x_i \in \mathbb{R}_{>0}, \tag{1.1}$$

$$f_2(x, y) := \frac{1}{|x-y|^{\mu}}, \quad x, y \in \mathbb{R}^d, \tag{1.2}$$

arising in various applications.

In this paper we describe and document an algorithm for solving the related nonlinear optimisation problem by an adaptive Newton-type method.

2 Newton Algorithm for Nonlinear Optimisation

We consider the unconstraint minimisation problem

$$\Phi(\mathbf{x}) \to min, \quad \mathbf{x} = (x_1, ..., x_N)^T \in \mathbb{R}^N \tag{2.1}$$

with a twice differentiable function $\Phi : \mathbb{R}^N \to \mathbb{R}$. Assuming that the local minimum $\mathbf{x}^*$ exists, we replace (2.1) by the solution of nonlinear system of equations

$$\mathbf{F}(\mathbf{x}^*) = 0 \quad \text{with } \mathbf{F} := \nabla\Phi : \mathbb{R}^N \to \mathbb{R}^N, \tag{2.2}$$

where $\nabla\Phi := (\frac{\partial\Phi}{\partial x_1}, ..., \frac{\partial\Phi}{\partial x_N})^T$ is the gradient of Φ. Given an initial guess $\mathbf{x}^0$, one step of the Newton method to solve (2.2) is given by

$$\mathbf{x}^{n+1} = \mathbf{x}^n - w_n \cdot \mathbf{J}^{-1}(\mathbf{x}^n)\mathbf{F}(\mathbf{x}^n), \quad n = 0, 1, ..., \tag{2.3}$$

where $\mathbf{J}(\mathbf{x}^n) = \{\mathbf{J}_{ij}\}$, $i, j = 1, ..., N$, is the Jacobian matrix for $\mathbf{F}$ calculated at the point $\mathbf{x}^n$, $\mathbf{J}_{ij} = \frac{\partial F_i}{\partial x_j}(\mathbf{x}^n)$ and $w_n \in (0, 1]$ is a relaxation parameter which has to be specified within the iteration process. The classical Newton method corresponds to $w_n = 1$. The Jacobian matrix $\mathbf{J}$ is, in fact, just the Hessian matrix of Φ. At each iterative step we solve the linear system of equations

$$\mathbf{J}(\mathbf{x}^n)\left(\mathbf{x}^{n+1} - \mathbf{x}^n\right) = -w_n \cdot \mathbf{F}(\mathbf{x}^n) \tag{2.4}$$

by the direct Gauss method.

For instance, we apply this procedure to find the best approximation of a certain real function $\phi : \Omega \to \mathbb{R}$, $\Omega \subset \mathbb{R}$ by exponential sums, that is

$$\Phi(\boldsymbol{\omega}, \boldsymbol{\alpha}) := \|\phi(x) - \sum_{\nu=1}^{N} \omega_\nu e^{-\alpha_\nu x}\|^2 \to min, \quad \mathbf{x} = (\boldsymbol{\omega}, \boldsymbol{\alpha}) \in \mathbb{R}^{2N} \tag{2.5}$$

with $\boldsymbol{\omega} = (\omega_1, ..., \omega_N)^T$, $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_N)^T$, and with $\omega_\nu, \alpha_\nu \in \mathbb{R}_+$, where $\|\cdot\|$ is some given norm on $\Omega = [1, R]$, $R > 1$. The most interesting choice of $\|\cdot\|$ corresponds to the $\|\cdot\|_{L^\infty(\Omega)}$-norm and to a weighted $L^2(\Omega)$-norm.

3 User Guide

3.1 General Description

For the implementation of the algorithm we use C++, where all calculations are done with `long double` precision, i. e., the machine accuracy is about 10^{-19} or 10^{-33} on a Pentium 4 or on a Sun Workstation, respectively. The program is tested under Linux and SunOS 5.7 with the `sunmath` library.

Input parameters (cf. Table 1) and coefficients can be changed within the C++ source code or within our program `newt` which runs in a terminal window. The first is more flexible the latter is easier. The function Φ, the vector function $\mathbf{F}$, and the Jacobian matrix $\mathbf{J}$ have to be specified on the source code level (see Figure 1).

Nearly all parameters can be modified during the iterative process given by equation (2.3). For controlling the relaxation parameter w_n we use the parameters w_{min} and w_{max} with $0 < w_{min} \leq w_n \leq w_{max} \leq 1$ $(n = 0, 1, \dots)$. A summary of all parameters and their default values used by the Newton and Gauss methods are given in Table 1.

If the n-th step of the Newton iteration satisfies certain convergence criteria, the relaxation parameter w_{n+1} is set to $\min\{w_{max}, 2\,w_n\}$ or else to $w_n/2$. It is also possible to change this parameter manually after each Newton step. If $w_n < w_{min}$ the iteration process aborts. For

Top level `newt`

Working with a certain optimisation problem, i.e. Φ, $\mathbf{F}$, and $\mathbf{J}$ are given (but the Newton method does not require Φ explicitly).
Loading and saving of the coefficient vector $\mathbf{x}$ and modifying of parameters.
Performing the Newton iteration given by equation (2.3).

Source code level C++

Specification of the functional Φ inside the member function `getPhi(vect& xn)` of class `Newton`, of left hand side $\mathbf{F}$ of equation (2.2) inside `getF(vect& xn)`, and of Jacobian matrix $\mathbf{J}$ inside `getJacobi(vect& xn)`.

Figure 1: Relationship between program `newt` and C++ source code. Notice that `vect` is equivalent to `vector<long double>`.

additional control of whether to accept $\mathbf{x}^{n+1}$ or not we use the parameter $q \in (0,1]$ by testing $\Phi(\mathbf{x}^{n+1}) < q\,\Phi(\mathbf{x}^n)$.

The Gauss elimination with pivoting used for solving equation (2.4) is controlled by the parameters piv_0 and piv_1. If the absolute value of the pivot element lies in $[0, piv_0)$ the program reports an error message and the iteration process aborts. If the pivot element lies in $[piv_0, piv_1)$ the program reports a warning message but the Newton process continues.

Parameter	`newt` key word	Range	Default value	C++ source code
w_{min}	`wmin`	$(0, w_{max}]$	10^{-3}	`setwmin(ldbl)`, `getwmin()`
w_{max}	`wmax`	$[w_{min}, 1]$	1	`setwmax(ldbl)`, `getwmax()`
w_0	`w0`	$[w_{min}, w_{max}]$	w_{max}	`setw0(ldbl)`, `getw0()`
w_n	`w`	$[w_{min}, w_{max}]$	–	`setw(ldbl)`, `getw()`
n (read-only)	`stepno`	$0, \ldots, n_{max}$	–	`stepno()`
n_{max}	`nmax`	$1, 2, \ldots$	20	`setnmax(ldbl)`, `getnmax()`
q	`qphi`	$(0, 1]$	1	`setq(ldbl)`, `getq()`
piv_0	`piv0`	$piv_0 > 0$	`LDBL_EPSILON`	`pivotctrl(ldbl, ldbl)` `getpiv0()`
piv_1	`piv1`	$piv_1 \geq piv_0$	$2\,piv_0$	`pivotctrl(ldbl, ldbl)` `getpiv1()`

Table 1: Summary of the parameters and their default values used by Gauss elimination and Newton method. `LDBL_EPSILON` is a C/C++ specific macro which corresponds to the machine accuracy. All functions given in the last column are member functions of the class `Newton` (see "./src/Newton.cpp"). Notice that `ldbl` is equivalent to `long double`.

3.1.1 Top level `newt`

Example 1: $x^2 + y^2 = 1$, $x + y = a$

Assuming that the directory containing `newt` is the working directory, then the program can be started by typing `newt` at the prompt of a terminal window (here the terminal prompt is indicated by the symbol `$`). The following session can also be started from the text file *example1.nwt* by typing `newt example1.nwt` at the terminal prompt.

```
$ newt

  Approximation by exponential sums 1.2 - Type ? or help.
  Example 1 :  x^2 + y^2 = 1, x + y = a

> example1                     # start Example 1
  x[0]^2 + x[1]^2 = 1 ,  x[0] + x[1] = a
  x[0] = 5.00000e-01 ,  x[1] = 0.00000e+00 ,  a = 1.41421e+00
  Parameters for Example 1 loaded. Press 'q' to leave Example 1.
> wmin = 0.1                   # set minimal stepsize to 0.1
> sci                          # set numerical output format to scientific
> prec=3                       # show three decimal places
> hide(gauss)                  # do not show Gauss elimination error
> a                            # show parameter a
  a = 1.414e+00
> a = 0.5                      # a = 0.5 -> QUADRATIC convergence
> x                            # show vector x
  x[0] = 5.000e-01
  x[1] = 0.000e+00
> store(1)                     # store actual configuration for later usage
  Stored at position 1.
> start                        # start calculation
  Step 0 (1): w = 5.000e-01 ,  || f || = 7.500e-01 ,  x = (5.000e-01, 0.000e+00)
> n=6                          # perform 6 steps of Newton iteration
  Step 1 (2): w = 1.000e+00 ,  || f || = 9.375e-02 ,  x = (8.750e-01, -3.750e-01)
  Step 2 (3): w = 1.000e+00 ,  || f || = 2.813e-03 ,  x = (9.125e-01, -4.125e-01)
  Step 3 (4): w = 1.000e+00 ,  || f || = 2.253e-06 ,  x = (9.114e-01, -4.114e-01)
  Step 4 (5): w = 1.000e+00 ,  || f || = 1.450e-12 ,  x = (9.114e-01, -4.114e-01)
  Step 5 (6): w = 1.000e+00 ,  || f || = 6.061e-20 ,  x = (9.114e-01, -4.114e-01)
  Terminated...
  0.000 sec
> restore(1)                   # load previous configuration
> a = 1.41421356237            # a = sqrt(2) -> LINEAR convergence
> start                        # restart calculation
  Step 1 (1): w = 1.000e+00 ,  || f || = 5.895e-01 ,  x = (1.250e+00, 1.642e-01)
> n=6                          # perform 6 steps of Newton iteration
  Step 2 (2): w = 1.000e+00 ,  || f || = 1.474e-01 ,  x = (9.786e-01, 4.357e-01)
  Step 3 (3): w = 1.000e+00 ,  || f || = 3.684e-02 ,  x = (8.428e-01, 5.714e-01)
  Step 4 (4): w = 1.000e+00 ,  || f || = 9.210e-03 ,  x = (7.750e-01, 6.392e-01)
  Step 5 (5): w = 1.000e+00 ,  || f || = 2.303e-03 ,  x = (7.410e-01, 6.732e-01)
  Step 6 (6): w = 1.000e+00 ,  || f || = 5.757e-04 ,  x = (7.241e-01, 6.901e-01)
  Step 7 (7): w = 1.000e+00 ,  || f || = 1.439e-04 ,  x = (7.156e-01, 6.986e-01)
> restore(1)                   # restore configuration
> a = 2                        # a > sqrt(2) -> NO convergence
> start
  Step 1 (1): w = 1.000e+00 ,  || f || = 1.125e+00 ,  x = (1.250e+00, 7.500e-01)
> n=6
```

```
   Step 1 (2): w = 5.000e-01 ,  || f || = 1.125e+00 ,  x = (1.250e+00, 7.500e-01)
   Step 1 (3): w = 2.500e-01 ,  || f || = 1.125e+00 ,  x = (1.250e+00, 7.500e-01)
   Step 2 (4): w = 5.000e-01 ,  || f || = 1.002e+00 ,  x = (9.688e-01, 1.031e+00)
   Step 2 (5): w = 2.500e-01 ,  || f || = 1.002e+00 ,  x = (9.688e-01, 1.031e+00)
   Step 2 (6): w = 1.250e-01 ,  || f || = 1.002e+00 ,  x = (9.688e-01, 1.031e+00)
   Step 2 (7): w = 6.250e-02 ,  || f || = 1.002e+00 ,  x = (9.688e-01, 1.031e+00)
   Abortion: w < wmin
   0.000 sec
 > # type 'q' to exit this example or 'qq' to exit program newt
```

To load the vector **x** of dimension 2 from file "./x0.dat" the assignment `x='x0.dat'` can be used. The file format has to be as follows:

```
0.5        # first vector component x[0]
0.0        # second component x[1]
```

3.1.2 C++ source code level

The core of the implementation is the base class `Newton` (see file "./src/Newton.cpp"). The parameters and corresponding member functions of this class are listed in Figure 1 and Table 1 above. Other important member functions are summarised in Table 2.

`Newton`		Description
`init()`	○	Initialisation of parameters after restart.
`init(vect& x0)`	○	Same as `init()`, but with specification of start vector.
`getx()`	–	Returns the vector of coefficients **x**.
`getxi(int i)`	–	Returns the `i`-th coefficient of vector **x**.
`setx(vect& xn)`	–	Sets the vector of coefficients **x** to `xn`.
`getxi(ldbl xi,`	–	Sets the `i`-th coefficient of vector **x** to `xi`.
`int i)`	–	
`init(vect& x0)`	○	Same as `init()`, but with specification of start vector.
`nextStep()`	–	Performs one step of Newton iteration.
`info()`	○	Output at the `newt` prompt after each Newton step.
`valid(vect& xn)`	○	Validates vector `xn` (used on initialisation and after each Newton step).
`test(vect& xn)`	○	Customises the test $\Phi(\mathbf{x}^{n+1}) < q\,\Phi(\mathbf{x}^n)$ in the Newton method.
`terminated(ldbl v)`	○	Returns `true` if certain condition is satisfied, else `false`. By default it tests if the Euclidean norm of **F** is less than `v`.
`loadFromFile(...)`	○	Reads vector **x** and parameters from file.
`saveToFile(...)`	○	Writes vector **x** and parameters to file.
`Description()`	●	Short description of the actual implementation that appears after startup of `newt`.
`PlotTitle()`	○	Returns a string that appears as the title of the plot window.
`PlotValue(ldbl)`	●	Returns a value for the plot command of `newt` (see below).

Table 2: Additional member functions of class `Newton`. The symbol ● indicates that the function has to be defined in all derived classes, whereas ○ means that this function has a default return value but can be overloaded (for informations about the return values see "./src/Newton.cpp").

A new optimisation problem is implemented by a derived class of the base class `Newton`. At first we create a new file with an arbitrary name and suffix `.cpp` (all source code files of the user implementation have to be saved into directory "./user/"). The name of the derived class is also arbitrary but has to be different from `Newton`. If we choose `NewtonExample1` we obtain the following class definition:

```
class NewtonExample1 : public Newton {
private:
  ldbl a;
public:
  NewtonExample1 ();                 /* constructor (parameter initialisation) */
  vect  getF      (const vect& xn);   /* vector function F    */
  MatNN getJacobi (const vect& xn);   /* Jacobian matrix of F */
  ldbl  geta ();
  void  seta (ldbl p);
  void  info ();                   /* output information after each Newton step */
  bool  saveToFile (string fnm, string rev, string host);
};
```

¿From outside the parameter a can be read with the member function `geta()` and modified with `seta(ldbl)`.

```
void NewtonExample1::seta (ldbl p) {
  /* test for validity of parameter a if necessary ... */
  a = p;
}

ldbl NewtonExample1::geta () {
  return a;
}
```

The constructor for the class `NewtonExample1` is necessary for initialisation of all parameters when creating a new object. For this example we want to set $a := \sqrt{2}$ by default, so we write

```
NewtonExample1::NewtonExample1 () : a (sqrtl(2.0)) { }
```

To print special information after each Newton step the member function `info()` can be overloaded. For an output of the form

```
Step 1 (1): w = 1.000e+00 ,  || f || = 1.125e+00 ,  x = (1.250e+00, 7.500e-01)
```

the function `info()` can be defined as follows:

```
void NewtonExample1::info () {
  cout << "  Step " << stepno () << " (" << Stepno () << "): w = " << w;
  cout << " ,  || f || = " << test (getF (getx ()));
  cout << " ,  x = (" << getxi (0) << ", " << getxi (1) << ")" << endl;
}
```

To complete the class definition we next write the member functions `getF(vect)` and `getJacobi(vect)` corresponding to our example :

```
vect NewtonExample1::getF (const vect& xn) {
  vect rv;     /* empty vector of long doubles */

  /* add first compoment x^2 + y^2 - 1.0 */
  rv.push_back (powl(xn[0], 2.0) + powl(xn[1], 2.0) - 1.0);

  /* add second component x + y - a */
  rv.push_back (xn[0] + xn[1] - a);

  return rv;
}

MatNN NewtonExample1::getJacobi (const vect& xn) {
  MatNN J (2);     /* square matrix of size 2 */

  J[0][0] = 2.0 * xn[0];  J[0][1] = 2.0 * xn[1];
  J[1][0] = 1.0;          J[1][1] = 1.0;

  return J;
}
```

The example in section 3.2.1 demonstrates how to create the N-dimensional function vector $\mathbf{F}$ and the square matrix $\mathbf{J}$ of size N by a loop statement.

3.1.3 Interface between top level `newt` and C++ source code

To make the parameter a visible on the top level `newt`, the C++ function `UserCommand(...)` has to be adapted ("./user/command.cpp"). This routine has the return value `true` if the command `cmd` is valid, otherwise it returns `false`. To avoid conflicts between parameter names it is always a good choice to start with an empty routine:

```
bool UserCommand (string cmd, string arg, string quotarg) {
  /* a return value of 'false' indicates that no command within UserCom-
     mand(...) was performed, otherwise the return value is 'true'; the
     return value serves only as indicator for syntax errors */

  return false;
}
```

For our example we want to handle two additional `newt` commands. The user input `a` should output the value of a. The user input `a=0.5`, for example, should ensure that parameter a has the value 0.5.

```
bool UserCommand (string cmd, string arg, string quotarg) {
  long double dblarg;

    /* see note in function Command(...), file ./src/command.cpp */

    /* print parameter value of a */
    if (cmd == "a")  { cout << "  a = " << Nwt.geta () << endl; return true; }

    /* change parameter value of a */
    if (cmd.find ("a=") != string::npos) {
      cmd.erase (0, 2);
```

```
        if (sscanf (cmd.c_str (), "%Lf\n", &dblarg) < 1) {
          cout << "  Parameter not changed (a = " << Nwt.geta () << ")." << endl;
        } else {
          Nwt.seta (dblarg);
        }
        return true;
      }
    return false;
  }
```

3.1.4 Source code compilation

Before compiling a new version of `newt` we have to adjust the file "./user/user.cpp". Assuming "./user/Example1.cpp" is the name of the file that contains the definiton of class `NewtonExample1` we write `#include "Example1.cpp"` instead of the previous `#include` directive in the special labeled section of file "./user/user.cpp". In the following line the `typedef` statement has to be replaced by

```
    typedef NewtonExample1 _Newton_;
```

where `NewtonExample1` is the name of the derived class. At this point the "./user/" directory contains the three files "user.cpp", "Example1.cpp", and "command.cpp". After deleting the old version of `newt` and typing `make` at the terminal prompt, the new program is available. Now it operates with the optimisation problem specified by `getF(vect&)`, `getJacobi(vect&)` and, possibly, `getPhi(vect&)` by default.

Some additional information about the source code can be found in the file "./README". This file contains also a reference to the source code documentation in HMTL format.

3.2 Implementations for the Case of Exponential Fitting

3.2.1 Example 2: Approximation to $1/x$

This example corresponds to the separable approximation of the function f_1 in (1.1) with the corresponding notation $N = d$.

Case 1. Function given by a "quadrature sum".

$$\mathbf{\Phi}(R;\boldsymbol{\omega},\boldsymbol{\alpha}) := \sum_{i=1}^{M} \frac{h_i}{2}\left(\Delta_i + \Delta_{i+1}\right), \quad \Delta_j := \left(\frac{1}{x_j} - \sum_{i=1}^{N} \omega_i e^{-\alpha_i x_j}\right)^2, \quad j = 1, ..., M+1,$$

with $\mathbf{x} = (\boldsymbol{\omega},\boldsymbol{\alpha}) \in \mathbb{R}^{2N}$, some $h_i > 0$, $i = 1, ..., M$ and $x_j \in [1, R]$, $j = 1, ..., M+1$, where $x_j = 1 + \sum_{k=1}^{j-1} h_k$. This quadrature sum is the well known trapezoidal rule for numerical integration corresponding to the L^2-norm functional F specified in Case 2 below.

The relevant parameters and corresponding key words and function names for this implementation are listed in Table 3. The first step of Newton iteration is performed with a constant interval width $h_i := h_{min}$, $i = 1, ..., M$. After $\mathbf{x}^1$ is computed, the interval width h_i is calculated by

$$h_i = \frac{12\varepsilon_Q}{g''(x_i + h_{min})} \ , \quad g(x) = \frac{1}{x} - \sum_{j=1}^{N} \omega_j e^{-\alpha_j x} \ , \quad i = 1, ..., M$$

with a parameter ε_Q which has to be specified manually (cf. Table 3).

For an example session with program `newt` we chose $N = 5$ and $R = 200$. The text file containing the start vector $\mathbf{x}^0$ has the following format, where additional lines of comment are allowed:

```
0.02199241319929076437901332118085573341666   {omega[1]}
```

```
0.0219924131992907643790133211808557334166   {omega[1]}
0.1002064224819224335166702351263001702364   {omega[2]}
0.3489637351854245363700929988270971193742   {omega[3]}
1.0398862719837947781158921101152259325318   {omega[4]}
2.9648211490348502911412048588246648250788   {omega[5]}
0.0077919805414365443251355311960609784094   {alpha[1]}
0.0610302875027291444151751869523492288749   {alpha[2]}
0.2635451761362904776547170376810313996430   {alpha[3]}
0.9023059551184773100754483998731103611135   {alpha[4]}
2.7287535886135676362583557530427924575633   {alpha[5]}
```

Parameter	`newt` key word	Range	Default value	C++ source code
R	`R`	$R > 1$	10.0	`setR(ldbl)` `getR()`
h_{min}	`hmin`	$($`LDBL_EPSILON`$, h_{max}]$	0.25	`sethmin (ldbl)` `gethmin ()`
h_{max}	`hmax`	$h_{max} \geq h_{min}$	h_{min}	`sethmax(ldbl)` `gethmax()`
h_i	–	$[h_{min}, h_{max}]$	–	–
ε_Q	`Qeps`	(0, 1]	10^{-2}	`setQeps(ldbl)` `getQeps()`
M (write-only)	M	1, 2, ...	–	`setM (ldbl)`

Table 3: Summary of additional parameters and their default values for example 2. The parameters for controlling the interval width of the quadrature (h_{min} and h_{max}) and ε_Q are only available for case 1. The parameter M is not implemented explicitly. It is used for setting $h_{min} = h_{max} = (R - 1)/M$.

The example session given below indicates a difference between the adaptive and constant width of the quadrature interval. This session can be obtained from file "newton1_x.nwt" (examples directory).

```
$ newt examples/newton1_x.nwt

  Approximation by exponential sums 1.2 - Type ? or help.
  Approximation of 1 / x in interval 1..R (quadrature sum).

> inputdir='1_x'          # set input directory  -> x='...'
> outputdir='data'        # set output directory -> save()
> x='1_xk05_2E2'          # load start vector from file
> prec=4                  # control output format
> nmax=30                 # maximal number of Newton steps
> R=200                   # set parameter R to 200
> store (1)               # store actual configuration for later usage
  Stored at position 1.
> hmin = 0.25             # ADAPTIVE width of quadrature interval
> hmax = 5
> Qeps = 1e-4             # set parameter Qeps to 0.0001
> s                       # start calculation
  Set quadrature intervals h := hmin = const.
  Step  1 ( 1): w = 1.0000e+00 ,  Phi   = 1.7862e-04 ,  ||f|| = 1.1526e-03
  Quadrature intervals are now adapted according to Qeps.
> n=20                    # 20 steps of Newton iteration
  Step  2 ( 2): w = 1.0000e+00 ,  Phi   = 1.7101e-04 ,  ||f|| = 9.0331e-04
  Step  3 ( 3): w = 1.0000e+00 ,  Phi   = 1.6136e-04 ,  ||f|| = 1.1973e-04
  ...
  Step 19 (19): w = 1.0000e+00 ,  Phi   = 1.5955e-04 ,  ||f|| = 3.1252e-14
  Step 20 (20): w = 1.0000e+00 ,  Phi   = 1.5955e-04 ,  ||f|| = 9.2033e-15
  Step 20 (21): w = 5.0000e-01 ,  Phi   = 1.5955e-04 ,  ||f|| = 9.2033e-15
> restore(1)              # restore first configuration
> M = 600                 # CONSTANT width of quadrature interval
  hmin = hmax = 3.3167e-01
> s                       # start calculation
  Set quadrature intervals h := hmin = const.
  Step  1 ( 1): w = 1.0000e+00 ,  Phi   = 1.6728e-04 ,  ||f|| = 1.1864e-03
  Equal quadrature intervals (hmin = hmax).
> n=20                    # 20 steps of Newton iteration
  Step  2 ( 2): w = 1.0000e+00 ,  Phi   = 1.6297e-04 ,  ||f|| = 6.4313e-04
  Step  3 ( 3): w = 1.0000e+00 ,  Phi   = 1.5328e-04 ,  ||f|| = 8.5992e-05
  ...
  Step 14 (14): w = 1.0000e+00 ,  Phi   = 1.5166e-04 ,  ||f|| = 4.7702e-14
  Step 15 (15): w = 1.0000e+00 ,  Phi   = 1.5166e-04 ,  ||f|| = 8.6922e-15
  ...
  Step 18 (21): w = 5.0000e-01 ,  Phi   = 1.5166e-04 ,  ||f|| = 1.0025e-15
> save('1_xk05_2E2') # write vector x into specified file
```

For calculating the functional $\mathbf{\Phi}$ at point $\mathbf{x}$ we use the subroutine `ldbl delta(vect& xn, ldbl x)` which returns

$$\texttt{delta} := \frac{\texttt{1}}{\texttt{x}} - \sum_{\texttt{i=0}}^{\texttt{N-1}} \texttt{xn[i] * exp (-xn[i+N]} * \texttt{x)}\ , \quad \texttt{x} \in [1, R],$$

where `xn` means vector $\mathbf{x}$. For the definition of the function `getPhi(vect)` we obtain

```
ldbl Newton1_x::getPhi (vect& xn) {
  ldbl rv = 0.0;

  /* set adaptive width of quadrature interval if necessary, i.e. after *
   * the first Newton step and if hmin is not equal to hmax             */
  seth ();

  int i, n = hvec.size ();

  /* hvec is the vector of interval widths h_i,         i=0,...,n-1  *
   * Hvec is the vector of corresponding values of x_i, i=0,...,n    *
```

```
 * x_i = 1 + h_0 + h_1 + ... + h_{i-1}                             */

/* Trapezoidal rule for integral approximation */
for (i = 0; i < n; i++) {
  rv += hvec[i] * (SQR (delta (xn, Hvec[i])) + SQR (delta (xn, Hvec[i+1])));
}
return 0.5 * rv;
}
```

In a similar way we implement `vect getF(vect)`:

```
vect Newton1_x::getF (vect& xn) {
  vect rv; ldbl sv, sv2;
  seth (); /* adaptive width of quadrature interval if necessary */
  int i, j, n = hvec.size ();

  for (i = 0; i < N; i++) {      /* derivatives with respect to omega */
    sv = 0.0;
    for (j = 0; j < n; j++) {
      sv -= hvec[j] * (delta (xn, Hvec[j])   * expl (-xn[i+N]*Hvec[j]));
      sv -= hvec[j] * (delta (xn, Hvec[j+1]) * expl (-xn[i*N]*Hvec[j+1]));
    }
    rv.push_back (sv);
  }

  for (i = 0; i < N; i++) {      /* derivatives with respect to alpha */
    sv = 0.0;
    for (j = 0; j < n; j++) {
      sv2  = delta (xn, Hvec[j])   * xn[i] * Hvec[j]   * expl (-xn[i+N]*Hvec[j]);
      sv2 += delta (xn, Hvec[j+1]) * xn[i] * Hvec[j+1] * expl (-xn[i+N]*Hvec[j+1]);
      sv += hvec[j] * sv2;
    }
    rv.push_back (sv);
  }
  return rv;
}
```

Parts of the routine `getJacobi(vect)` are listed below. The full source code can be found in file "./user-1_x/Newton1_x.cpp".

```
MatNN Newton1_x::getJacobi (vect xn) {
  MatNN Jac (2 * N); ldbl sv, sv2;
  seth (); /* set adaptive width of quadrature interval if necessary */
  int i, j, k, n = hvec.size ();

  for (i = 0; i < N; i++) {     /* derivatives with respect to omega, ... */
    for (j = 0; j < N; j++) {   /* omega, omega */
      sv = 0.0;
      for (k = 0; k < n; k++) {
        sv2 = expl (-(xn[i+N] + xn[j+N])) * (Hvec[k] + Hvec[k+1]);
        sv += hvec[k] * sv2;
      }
      Jac[i][j] = sv;
    }
    ...                         /* omega, alpha */
  }
```

```
    ...                           /* derivatives with respect to alpha, ... */
    return Jac;
  }
```

Remark 1

The assignment `x='file.txt'` can be used to load a vector of coefficients from file "file.txt". To use a different file format than given on page 9, the C++ function `loadFromFile(...)` has to be adapted. The corresponding C++ function for the `newt` command `save('file.txt')` is `saveToFile(...)`. The standard input and output directory can be changed by setting the `newt` variables `inputdir` and `outputdir`, respectively:

```
> inputdir                        # the working directory, by default
  Input directory is '.'.
> inputdir='1_x'                  # set input directory to ./1_x
> inputdir                        # the input directory is now ./1_x
  Input directory is '1_x'.
> outputdir='data'                # set output directory to ./data
```

It is important to notice that existing files are overwritten without confirmation.

Remark 2

To visualise the difference between the exact value of the considered function and the approximation, the `newt` command `plot(...)` can be used. Some examples are

```
> plot(log, 1, 200)             # screen plot with logarithmic x-scale, range = 1..200
> plotdir
  Plot directory is '.'
> plotdir = 'plots'               # set output directory for plots to './plots/'
> plot('plotvalues.txt', 1, 200) # save raw data (note: no log/nolog argument!)
> plot('plot.eps', log, 1, 200)  # log/nolog argument indicates postscript
```

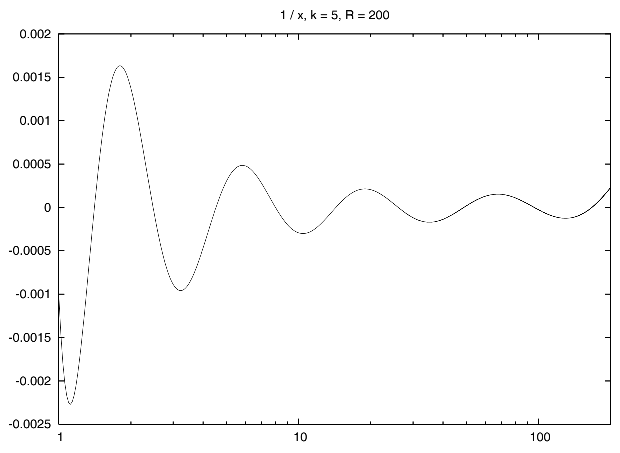

Figure 2: Plot generated with the command `plot(log, 1, 200)`.

The output of the `plot(...)` command depends on the definition of the C++ function `PlotValue(ldbl)` which can be redefined (overloaded) for every new optimisation problem.

Case 2. Explicitly given function.

Given $R > 1$, $N \geq 1$, find $2N$ real parameters $\alpha_1, \omega_1, ..., \alpha_N, \omega_N \in \mathbb{R}_{>0}$, such that

$$\begin{aligned} F(R; \boldsymbol{\omega}, \boldsymbol{\alpha}) :=& \int_1^R \left(\frac{1}{x} - \sum_{i=1}^N \omega_i e^{-\alpha_i x} \right)^2 dx \\ =& 1 - \frac{1}{R} - 2 \sum_{i=1}^N \omega_i \left[Ei(\alpha_i) - Ei(\alpha_i R) \right] + \frac{1}{2} \sum_{i=1}^N \frac{\omega_i^2}{\alpha_i} \left[e^{-2\alpha_i} - e^{-2\alpha_i R} \right] \\ &+ 2 \sum_{1 \leq i < j \leq N} \frac{\omega_i \omega_j}{\alpha_i + \alpha_j} \left[e^{-(\alpha_i + \alpha_j)} - e^{-(\alpha_i + \alpha_j) R} \right] \to \ min, \end{aligned} \tag{3.1}$$

where the exponential-integral function is given by

$$Ei(x) = \int_x^\infty \frac{e^{-t}}{t} dt, \quad x > 0.$$

In the special case $R = \infty$, we have

$$F(\infty; \boldsymbol{\omega}, \boldsymbol{\alpha}) := 1 - 2 \sum_{i=1}^N \omega_i Ei(\alpha_i) + \frac{1}{2} \sum_{i=1}^N \frac{\omega_i^2}{\alpha_i} e^{-2\alpha_i} + 2 \sum_{1 \leq i < j \leq N} \frac{\omega_i \omega_j}{\alpha_i + \alpha_j} e^{-(\alpha_i + \alpha_j)}.$$

The source code for this example can be found in directory "./user-1_x_explict/".

3.2.2 Example 3: Approximating $1/\sqrt{x}$ by Weighted Exponential Sums

This example corresponds to the separable approximation of the function f in (1.2).

Given $R > 1$, $N \geq 1$, find $2N$ real parameters $\alpha_1, \omega_1, ..., \alpha_N, \omega_N \in \mathbb{R}_{>0}$, such that

$$F_1(R; \alpha_1, \omega_1, ..., \alpha_N, \omega_N) := \int_1^R \frac{1}{x} \left(\frac{1}{\sqrt{x}} - \sum_{i=1}^N \omega_i e^{-\alpha_i x} \right)^2 dx \ \to \ min.$$

The functional is represented by the trapezoidal quadrature (cf. Example 1, Case 1). Then we apply the Newton method to solve the optimisation problem.

To keep the implementation of the function `getPhi(vect)` as most as simple we define four additional functions within the class `Newton1_sqrtx`:

```
/* returns coefficients of the first half, i.e. omega_i, i=0,...,N-1 */
ldbl Newton1_sqrtx::omega (vect xn, int i) {
  int N = xn.size () / 2;

  if ((i < 0) || (i >= 2 * N)) {
    cout << "  Error: Index out of range." << endl; return 0.0;
  }

  if (i < N) { return xn[i]; } else { return xn[i - N]; }
}
```

```
/* returns coefficients of the second half, i.e. alpha_i, i=N,...,2*N-1 */
ldbl Newton1_sqrtx::alpha (vect xn, int i) {
  int N = xn.size () / 2;

  if ((i < 0) || (i >= 2 * N)) {
    cout << "  Error: Index out of range." << endl; return 0.0;
  }

  if (i >= N) { return xn[i]; } else { return xn[i + N]; }
}

/* returns the value of the exponential sum according to vector xn */
ldbl Newton1_sqrtx::ExpSum (vect xn, ldbl xx) {
  ldbl rv = 0.0;
  int i, N = xn.size () / 2;

  for (i = 0; i < N; i++) { rv += omega (xn, i) * expl (-alpha (xn, i) * xx); }
  return rv;
}

/* returns the difference between real function value and approximation */
ldbl Newton1_sqrtx::I (vect xn, ldbl xx) {
  ldbl d = 1.0 / sqrtl (xx) - ExpSum (xn, xx);
  return 1.0 / xx * d * d;
}
```

These four functions make `getPhi()` easy to read:

```
/* trapezoidal approximation */
ldbl Newton1_sqrtx::getPhi (vect xm) {
  int i, ldbl rv = 0.0;

  for (i = 1; i < M; i++) { rv += I (xm, 1.0+i*h); }
  rv += 0.5 * I (xm, 1.0);
  rv += 0.5 * I (xm, R);
  return h * rv;
}
```

In a similar way some other functions are introduced to make the code listings for `getF()` and `getJacobi()` shorter:

```
vect Newton1_sqrtx::getF (vect xm) {
  vect rv;
  ldbl sv;
  int k, i, N = xm.size () / 2;

  /* derivatives with respect to omega */
  for (k = 0; k < N; k++) {
    sv = 0.0;
    for (i = 1; i < M; i++) { sv += DwI (xm, 1.0+h*i, k); }
    sv += 0.5 * DwI (xm, 1.0, k);
    sv += 0.5 * DwI (xm, R,   k);
    rv.push_back (h * sv);
  }

  /* derivatives with respect to alpha */
```

```
    for (k = 0; k < N; k++) {
      sv = 0.0;
      for (i = 1; i < M; i++) { sv += DaI (xm, 1.0+h*i, k); }
      sv += 0.5 * DaI (xm, 1.0, k);
      sv += 0.5 * DaI (xm, R,   k);
      rv.push_back (h * sv);
    }

    return rv;
  }
```

The two additional functions which returns the value of the derivatives with respekt to ω_k and α_k, $k = 0, 1, ..., N-1$ are defined as follows:

```
/* returns the derivative with respect to omega_k */
ldbl Newton1_sqrtx::DwI (vect xm, ldbl xx, int k) {
  return -2.0 / xx * expl(-alpha(xm,k)*xx) * (1.0 / sqrtl(xx) - ExpSum(xm, xx));
}

/* returns the derivative with respect to alpha_k */
ldbl Newton1_sqrtx::DaI (vect xm, ldbl xx, int k) {
  return 2.0 * omega(xm,k) * expl(-alpha(xm,k)*xx) * (1.0 / sqrtl(xx) - ExpSum(xm, xx));
}
```

The complete source code including the defintion of `getJacobi()` can be found in "./user-1_sqrtx/Newton1_sqrtx.cpp".

References

[1] D. Braess and W. Hackbusch: Approximation of $1/x$ by Exponential Sums in $[1, \infty)$. To appear in IMA J. Numer. Anal.

[2] I. P. Gavrilyuk, W. Hackbusch and B. N. Khoromskij: *Hierarchical Tensor-Product Approximation to the Inverse and Related Operators for High-Dimensional Elliptic Problems.* Computing 74 (2005), 131-157.

[3] L. Grasedyck: *Existence and computation of a low Kronecker-rank approximation to the solution of a tensor system with tensor right-hand side.* Computing 70 (2003), 121-165.

[4] W. Hackbusch and B.N. Khoromskij. *Hierarchical Kronecker Tensor-Product Approximation to a Class of Nonlocal Operators in High Dimensions.* Parts I/II. Preprints 29/30, MPI MIS, Leipzig, 2005; To appear in Computing.

[5] W. Hackbusch, B.N. Khoromskij and E. Tyrtyshnikov: *Hierarchical Kronecker tensor-product approximation.* Preprint 35, Max-Planck-Institut für Mathematik in den Naturwissenschaften, Leipzig, 2003; To appear in J. Numer. Anal.