The code on solving Fredholm equation of the first kind: usage instructions

I.I.Antokhin
Sternberg State Astronomical Institute, Lomonosov Moscow State University

September 26, 2023

1 General notes

1. While some code is implemented to prevent most evident user errors (like setting the initial approximation for `z` outside the region defined by the constraints etc.), the code is not fool proof. Making it such would take a lot of efforts and time which I do not have.

2. The code "as is" can be used for solving Fredholm equation with kernels appropriate for solving WR+O light curves, or for solving Abel equation. If you need other kernels for Fredholm, you should provide your own functions computing the kernels (see below).

2 Files in the package

```
prgrad_reg.c         - functions to solve Fredholm and Abel equations
wr_o_common.c        - equation kernels for Fredholm eqs. solving WR+O light curves
                       and Abel eq, also routines computing elliptical functions
test_prgrad.c        - test program showing how to run functions from prgrad_reg.c
Makefile             - make file to compile test_prgrad

test_compact.dat     - The output of the test program, solution on a compact set, no
                       Tikhonov's regularization
test_reg_W21.dat     - Same, on the same compact set plus Tikhonov's regularization
                       in the metric space W_2^1
test_reg_W22.dat     - Same, metric space W_2^2

prgrad_reg_usage.pdf - this file
```

3 Compiling

The code is developed in Linux and compiled with gcc 12.3.1. There should be no problem compiling it with any ANSI C compiler. In a non-Linux OS you will have to replace the random generator functions `srandom` and `random` with your local versions. These functions are used in the test code only, to add Gaussian noise to the simulated input data.

To compile the test program in Unix environment, run `make` (correct the supplied Makefile if needed). After compiling, run the test program as

`test_prgrad`

More on the test code and a simulated model used in it, below. Note that in UNIX/Linux lines of text files are ended with `<LF>` while in Windows with `<CR><LF>`. If you use Windows you may want to convert the files to `<CR><LF>` format, otherwise you will see a file as a single line.

4 The structure of the calling (main) program

It is linear and very simple. See comments in the source code `test_prgrad.c`. Outline:

```
Define necessary variables.
Read input data.
Calculate the uncertainty of input data and the sum of weights.
Compute the constraints matrix and its right-hand part.
Set initial approximation for z.
Call the main function solving the equation (ptizr_proj).
Calculate model Az.
Output the results.
```

Variables: In my code the memory for all arrays is allocated dynamically. If you prefer, you may set most of 1D arrays (u, z, az, grids, weights etc.) in the calling program as static. Exceptions are the constraints matrix `con` and its right-hand part `b` which are allocated within the function `ptilrb` and must be defined as pointers (see the source code in `test_prgrad.c` and comments below).

Read input data: For simplicity and readability, in `test_prgrad.c` I hard-coded reading them in the body of the `main` function. You may wish to write a separate function and call it from `main`. Same for setting algorithm parameters. I actually do this in my practical applications of the code.

Uncertainty of input data and the sum of weights: These are numeric representations of the uncertainty

$$\delta^2 = \int_c^d \sigma^2(x)dx$$

(where $\sigma(x)$ is uncertainty of $u(x)$) and the integral of weights

$$sumv = \int_c^d v(x)dx$$

where $v(x)$ is weight of $u(x)$, e.g. $\sim 1/\sigma^2(x)$. See the test code for further details.

5 Description of functions to call from your main program

5.1 Constraints matrix

A priori constrains have the form

$$con \cdot \vec{z} \leq \vec{b}$$

where con is the constraints matrix and $\vec{b}$ is the right-hand part. They are computed by

```
void ptilrb( int kernel_type, int switch_contype, int *n_con, double ***con,
             double **b, int n, double c2, int icore, int l, int *ierr );

Parameters:

kernel_type    - 1,2, or 3. 1 and 2 are kernels for Fredholm eq. when
                 solving light curves of WR+O binaries. For Abel equation,
```

```
                 you must set it to 3. You can add your own kernels and
                 enumerate them as 4, 5, ...
switch_contype - type of constraints on the unknown function:
                 = 1 - monotonically non-increasing and non-negative
                       (constraints 2,3,4 also include this constraint)
                 = 2 - concave z" <= 0
                 = 3 - convex  z" >= 0
                 = 4 - concave-convex with inflection point index l
                 = 5 - non-negative (this is to use Tikhonov's regularization
                       only without specifying any compact set)
n_con          - the number of rows in con, computed within the function
con            - constraints matrix. Note that in the calling program you must
                 declare it as "double **con". Memory for con is allocated within
                 the function. The cell values are also computed within the function.
b              - right-hand part of constraints. In the calling program must be
                 declared as "double *b". Allocated and computed within the function.
n              - dimension of z.
c2             - The value of z(0) (if known, see below). If unknown, set to
                 an arbitrary value. However, you will need it for initial
                 approximation, so may set to a reasonable value before caling
                 ptilrb (see below).
icore          - If it is known that z(s)=c2 at s<s0, where s0 is some known value,
                 icore sets the maximal index of a grid knot where, in numeric
                 representation of z, z[i<=icore]=c2 (see below). For Abel equation,
                 MUST be set to 0: icore=0.
l              - Index of inflection point for switch_contype=4. For other
                 constraint types, not used, set to an arbitrary value.
ierr           - return code. 0 if all is ok, 202 if memory for con or b could
                 not be allocated.
```

Note that when calling `ptilrb`, you must use `&con, &b` and not just `con, b`.

5.2 Initial approximation

```
void initialapprox(int switch_contype, double c2, double *s, double *z, int l, int n)

switch_contype - same as above
c2             - The value of z[0] of initial approximation. Should be roughly up
                 to the expected scale of z. E.g. if you expect z to vary between
                 0 and 100, set c2=20 or 50 etc. Do not set it to e.g. 1e10 or
                 1e-10 as this may cause numerical problems.
s              - grid on s - array of size n. Argument of z.
                 IMPORTANT: grid must be even (s[i+1]-s[i]=const)!!!
z              - Unknown function z. Array of size n.
l              - Index of the inflection point, see above.
n              - Size of grid on s. Dimension of s and z.
```

5.3 The ptizr_proj function

Here is the main function solving Fredholm or Abel equations. I provide a list of all its parameters and then give explanations on some of them.

```
void ptizr_proj( int kernel_type, int switch_contype, int n_con, double **con,
                 double *b, double rstar, double *u0, double *v, double sumv,
                 double *s, double *x, double xmin, double xmax, int n, int m,
                 double *z, double c2, double dl2, double eps, double h,
                 int adjust_alpha, double *alpha, char *metric, int l, int icore,
                 double ax, double *del2, int imax_reg, int *iter_reg,
                 int imax_minim, int *iter_minim, int verbose, int *ierr );

kernel_type    - Same as above.
switch_contype - Same as above.
n_con          - Same as above.
con            - Same as above.
b              - same as above.
rstar          - When solving a set of Fredholm eqs. for WR+O light curves, sets the
                 radius of the O star (used in kernels). In Abel equation, or if you
                 use your own kernels in Fredholm, set to an arbitrary value.
u0             - Input data, right-hand part of Fredholm eq. Array of size m.
                 The same as u from the paper.
v              - Weights of u0 data points.
sumv           - Sum of weights (see the code in test_prgrad.c).
s              - Same as above.
x              - Argument of u0. Array of size m.
xmin           - Lower limit of x.
xmax           - Upper limit of x.
n              - Dimension of z.
m              - Dimension of u0. The number of input data points.
z              - Unknown function. Array of size n. When calling the function, must
                 contain initial approximation. On return, contains the solution.
c2             - Same as above.
delta2         - Uncertainty of input data (see above).
eps            - Threshold for solving ||Az-u||^2-delta2=0. See below.
h              - If known, uncertainty of the A operator. The difference between
                 exact Az and its numeric approximation. At reasonably large n,
                 it is safe to set h=0.0.
adjust_alpha   - If 1, search for optimal regularization parameter. If 0, solve
                 at fixed alpha.
alpha          - Either alpha value to use or its initial value.
metric         - Metric space to use. May be "L2", "W21", or "W22".
l              - Index of the inflection point as above.
icore          - As above.
ax             - A parameter used when solving light curves of WR+O (needed for
                 computing the kernels. For your own kernels or in Abel eq.,
                 set to an arbitrary value.
del2           - Residual of the model relative to the input data.
imax_reg       - Max number of regularization iterations.
iter_reg       - Actual number of regularization iterations made.
imax_minim     - Max number of conjugate gradients minimization iterations made in
                 the last regularization iteration.
iter_minim     - Actual number of conjugate gr. iterations made.
verbose        - If set to 1, some additional stdout printing is done. Set to 0 to
                 skip verbose printing.
ierr           - Return code. 1** - normal end, 2** - various errors
                = 100 - normal end, exact minimum is found
                = 101 - iterations finished by residual value
```

```
= 102 - iterations finished by the norm of the gradient
= 103 - alpha became equal to zero while doing iterations
= 104 - when doing initial try at alpha=0, an4 >= eps*dl2,
        regularization impossible (see below)
= 200 - initial approximation outside allowable range.
= 201 - inconsistent adjust_alpha and alpha
= 202 - errors allocating memory for working arrays
= 203 - singular matrix of active constraints (when computing
        the projector)
= 204 - initial alpha < 0
= 205 - while searching for the interval on alpha containing
        the solution, made imax_reg multiplications by 2.0,
        residual still negative. Initial regularization parameter
        too small.
= 206 - max number of "big loop" (imax_minim) iterations reached,
        no solution.
= 207 - in bisection method, imax_reg iterations done, still not
        reached exit criteria.
= 208 - kernel_type <1 or >4.
```

Hopefully, the meaning of most of the parameters above is clear. A few comments on some of them:

Parameters controlling Tikhonov's regularization:

If `adjust_alpha=0` and `alpha=0`, no Tikhonov's regularization is done. The equation is solved on a compact set. If `adjust_alpha=0` and `alpha>0`, Tikhonov's regularization at this fixed value of `alpha` is performed. If `adjust_alpha=1`, a search for the optimal `alpha` is performed. Before calling `ptizr_proj` `alpha` must be set to some positive value, otherwise the function will return an error. This positive value is used as a starting point in searching for the optimal `alpha`.

Optimal `alpha` is defined by the condition

$$an4 \equiv ||Az - u||^2 - \delta^2 = 0$$

The function on the left is a monotonically increasing function of `alpha` (see Fig. 1). So the solution of the above equation is located between the points `alpha=0` and some `alpha_1` such that the left hand part of the above equation is positive. The search is performed as follows:

1. Compute `an4` at `alpha=0`. If `an4<0`, proceed, otherwise exit (see below).
2. Compute `an4` at `alpha` set before calling `ptizr_proj`.
3. If `an4` is negative, multiply `alpha` by 2 and repeat (2).
4. Repeat (3) until `an4` becomes positive (at some `alpha_1`).
5. Now, the solution is between `alpha=0` and `alpha=alpha_1`.
6. Use the bisection method to solve the above equation.

In practical terms, the solution is considered to be found if

$$||Az - u||^2 - \delta^2 < \epsilon\delta^2$$

Thus, variable `eps` sets the threshold for solution of the above equation.

There may be a situation when even at `alpha=0`

$$an4 > \epsilon\delta^2$$

. This may happen e.g. if a priori constraints do not allow the model to fit the data very well. Or, if the uncertainty of input data δ^2 is underestimated. No Tikhonov's regularization is possible in this case and the function exits with the errorcode `ierr=104`. z contains the solution at `alpha=0`.

Inflection point

If you want to use the concave-convex type of constraints, you must set the position (index) of the inflection point. If it is known, just set it once and call `ptizr_proj`. If not, you can search for it by repeatedly calling `ptizr_proj` with various values of `l` and choosing the optimal value by the minimum of `del2`. Clearly, `l>=1` and `l<=n-2` (recall that in `C` array indexes start from 0 so the first and last indexes of an array of the length n are 0 and n-1). Otherwise, `z` will be simply convex or concave. You can use these values of `l` as the end points of the possible `l` interval and find the optimal `l` by e.g. the golden section method. I have the code for the method but did not include it in the test code, as I wanted to keep it simple. In any case, it is easy to implement.

In reality the situation may be more complicated than that. For instance, in case of the simulated unknown function I included in this package, you can see (Fig.2 below) that in the range $s \sim 0.2 - 0.4$ the function is only slightly curved (not very different from a straight line). So if you choose various $s(l)$ to be between 0.2 and 0.4 the deviations from the data will be about equal. Only at $s(l) >\sim 0.6$ the concave part of model z will be very different from the true (exact) function and the deviation will become large. Moreover, as the central (small s) parts of z give relatively small contribution to the total value of Fredholm integral, at small `l` the deviations will increase not so much. This is illustrated in Fig.2. Note that the true value of `l` in this example (the inflection point of the exact simulated z) is equal to 40, corresponding to $s = 0.295$.

This illustration shows that it would be a bad idea to blindly choose optimal `l` by the minimum of the norm. The dependency of the norm on `l` is strongly defined by the shape of the unknown function. For simulated functions used in my examples, parts of which are nearly straight lines, there will be no clear "parabola"-like shape of the dependency unambiguously defining `l`. If real $z(s)$ is a curved function with the second derivative significantly different from zero at all points, the situation may be different.

On the other hand, if a part of your unknown function is nearly straight line, there is no much difference which value of `l` you choose. As long as its is located within the straight part of the function, the shape of the solutions with different `l`-s will be nearly identical. Still, I make these comments so that you will understand the possible behaviour of the algorithm.

In practical terms, this means that it is a good idea to run the program for all possible values of `l` and produce a plot like my Fig.2. If the plot shows well defined minimum,the corresponding value of `l` may be easily choosen. If not, you might use some independent considerations to choose the optimal `l`, e.g. if you have some quesses on possible shape of your unknown function.

One further note has to be made. If you want to use Tikhonov's regularization with the concave-convex constraint, do not use it when searching for the optimal `l`. Instead, while searching for such `l`, solve your equation on a compact set. The reason is that with Tikhonov's regularization switched on, you will always get del2 = δ^2 and will not be able to choose the optimal `l` by the minimum of `del2`. Also, at different `l` the values of optimal `alpha` will be different and the corresponding solutions are hard to compare. So search for the optimal `l` not using Tikhonov's regularization, and then run `ptizr_proj` once more with the optimal value of `l` (just found) AND Tikhonov's regularization switched on.

icore and c2

Whatever is the kernel of Fredholm equation, `c2` is used to set `z[0]` in the initial approximation of `z`. However, in my WR+O code it also has another use. In this code, I solve not just one Fredholm equation but a system of three eqs., two Fredholm ones and one algebraic eq. setting the normalization. After solving the first Fredholm eq., by using the normalization equation, I get `c2` such that in the second Fredholm equation $z(0)$ must be equal to `c2`. Moreover, in that latter equation $z(s)$ is the opacity of the WR disk $z(s) = c2(1 - e^{-\tau(s)})$, where $\tau(s)$ is the optical depth at the impact distance s from the disk center. Clearly, even though a WR star has semi-transparent wind, it also has a completely non-transparent core $\tau(s < rcore) = \infty$. So $z(s < rcore) = c2$. `icore` is computed as rcore (set in the input file) divided by ds, the grid step size. Thus, `icore` is the index of the last point of the grid on `s`, where

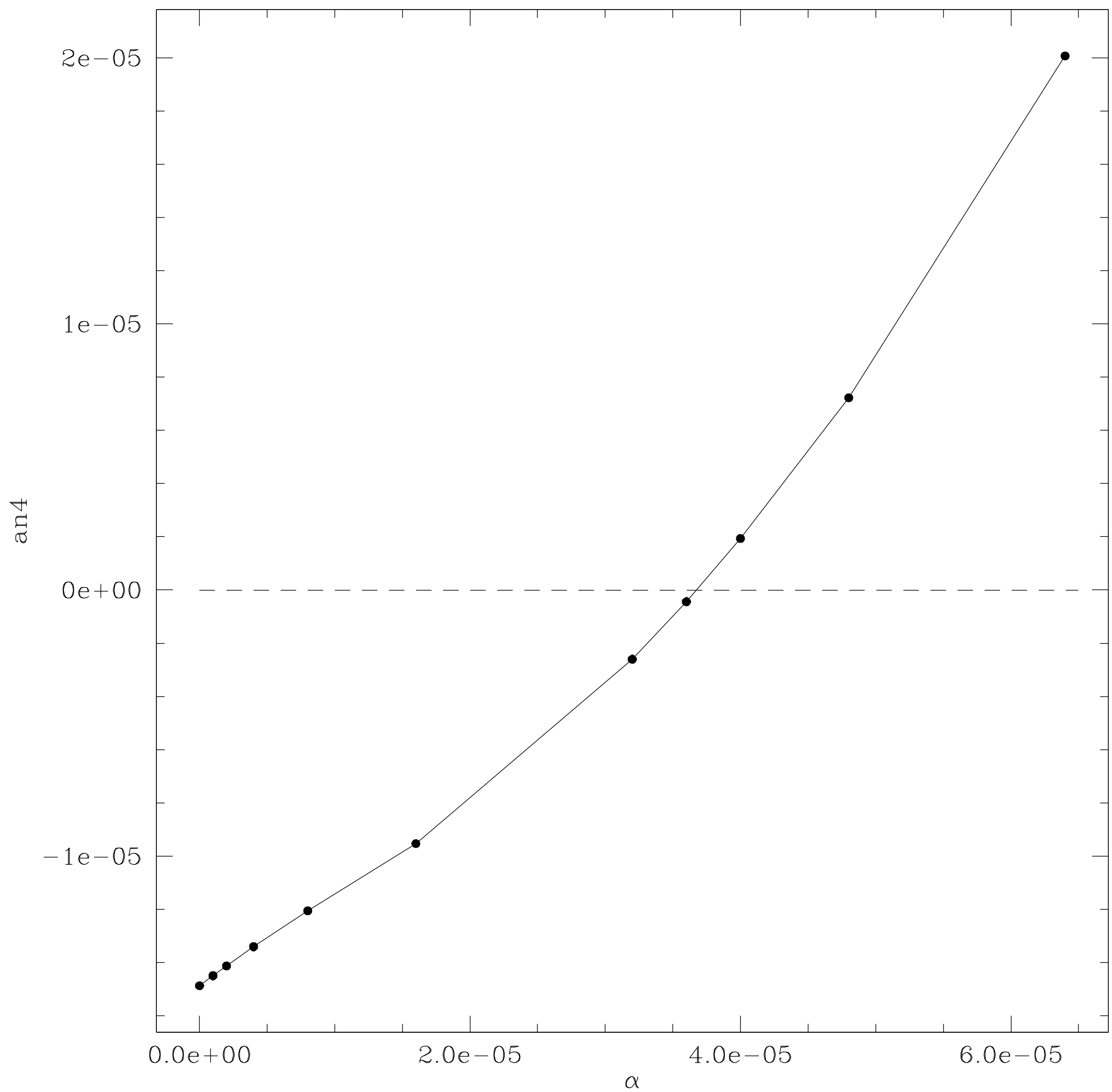

Figure 1: Dependence of an4 from α. Black dots show the sequencial steps in bisection search for the optimal α.

z[i] = c2. Important: if icore>0, the unknown function z(i) is fixed to c2 at all grid knots with indexes from 0 to icore.

This means that if you do not have such constraint on your unknown function, set c2 to some reasonable value (see above) before calling initialapprox and set icore=0. If icore>0, your solution $z(s)$ will be fixed to c2 at all z[0<=i<=icore].

5.4 Computing model u (Az)

This is simply calculating the Fredholm integral with model z. To do this, first compute the kernel (of course, it is computed in ptizr_proj, but it is a local variable not seen outside ptizr_proj).

```
Allocate memory for the kernel. Function "matrix'" allocates memory
for a 2D array with m rows and n columns,

double **a = matrix( int m, int n )

Compute Az (the model function to compare with the input data):

void pticr3( double **a, double *z, double *az, int n, int m )
```

6 The test program

In the test program I tried to keep the code to bare essentials for readability and easier understanding. All parameters and data are hard-coded in the "main" function. All output goes to stdout. Redirect it to a file if you wish.

The test program solves the Fredholm equation

$$Az = \int_{a}^{b} K(x,s)z(s)\,\mathrm{d}s = u(x), \quad s \in [a,b], x \in [c,d]$$

The simulated model is defined as
the kernel:

$$K(x,s) = \frac{1}{1+100(s-x)^2}$$

The intervals $[a,b] = [0,1]$, $[c,d] = [0,1]$. The exact z is set to be a concave-convex function

$$z_0(s) = \frac{3}{2}\left(\cos(\pi\frac{s}{2s_l}) + 1\right)$$

so that $z_0(0) = 3$. The position of the inflection point $s_l = 0.4$. With the given kernel and z_0, the integral in the left hand side is computed, resulting in the exact right hand part $u_0(x)$. Gaussian noise with $\sigma = 0.01$ is then added to $u_0(x)$ resulting in $u(x)$ which is used as input data for test solutions.

The test solutions are provided for three cases:

1. Solution on a compact set of non-negative, monotonically non-increasing concave-convex functions without Tikhonov's regularization.
2. The same, with added Tikhonov's regularization in the metric space W_2^1.
3. The same, with added Tikhonov's regularization in the metric space W_2^2.

The results are shown in Fig.3-5. To get a feeling on how the algorithm works, you may wish to play with the test model e.g. by changing the value of Gaussian noise, kernel parameters, using different constraint types (e.g. switch_contype=1 for solving the equation on a set of monotonic functions) etc.

7 Adding your own kernel

The kernel of the Fredholm's equation is computed by the `pticr0` function in `prgrad_reg.c`. However, since I have more than one kernel and in every one there is some specifics, `ptirc0` does not include the code computing the various kernels, directly. Instead, it calls functions which do this for various cases. These functions for the 4 current kernels (WR+O light curve eclipses 1 and 2, Abel equation, and the test problem) are locates in `wr_o_common.c`. To use your own kernel, you may add kernel number 5 by adding a call to your own function from `pticr0` and adding the function itself to `wr_o_common.c`.

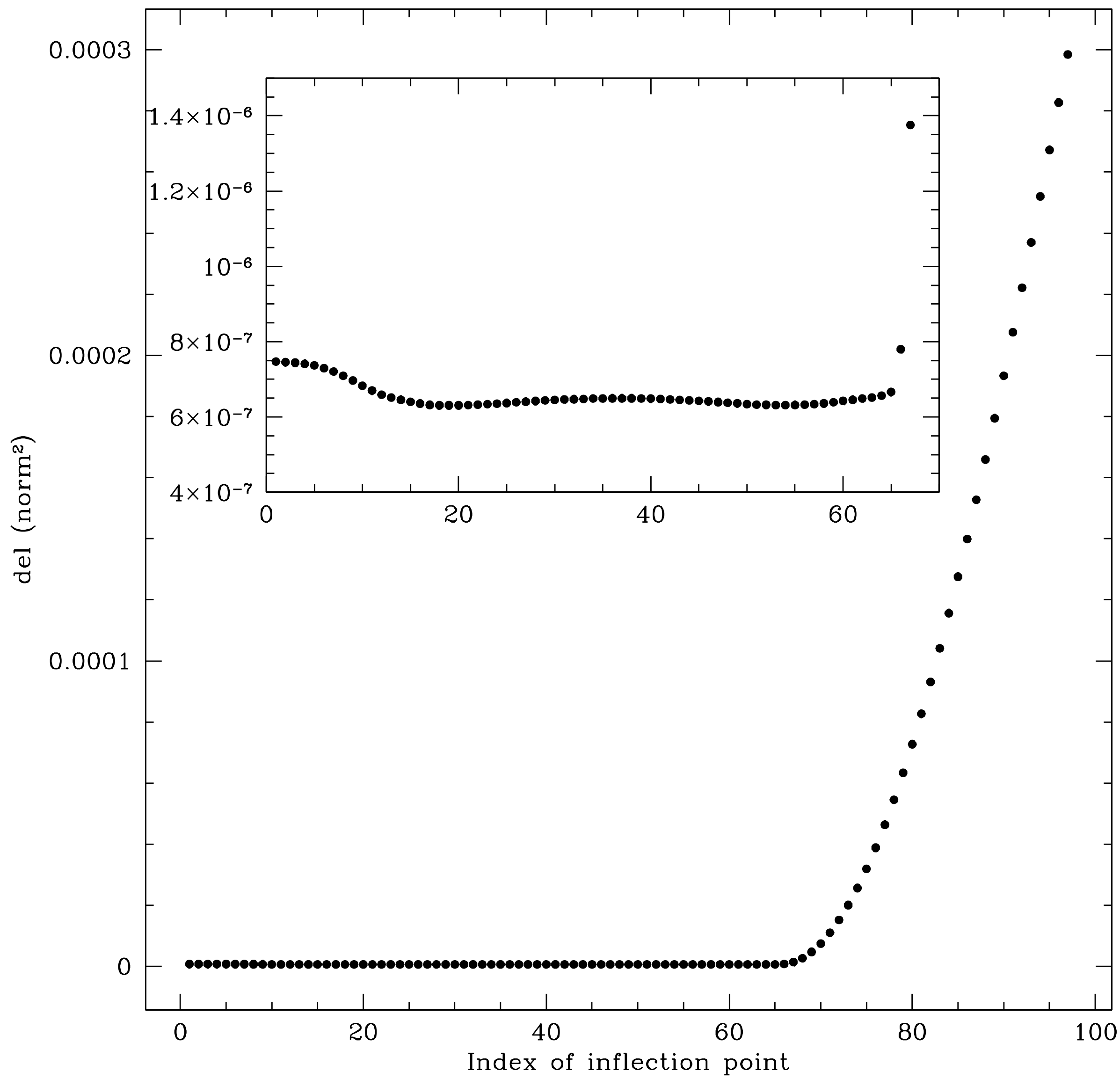

Figure 2: Dependency of the norm $||Az - u||^2$ from the position of the inflection point l, for a simulated function with known solution, with added Gaussian noise (sigma=0.001) and without Tikhonov's regularization. See comments in text. In the insert, the lower-left part of the plot is shown at an increased scale.

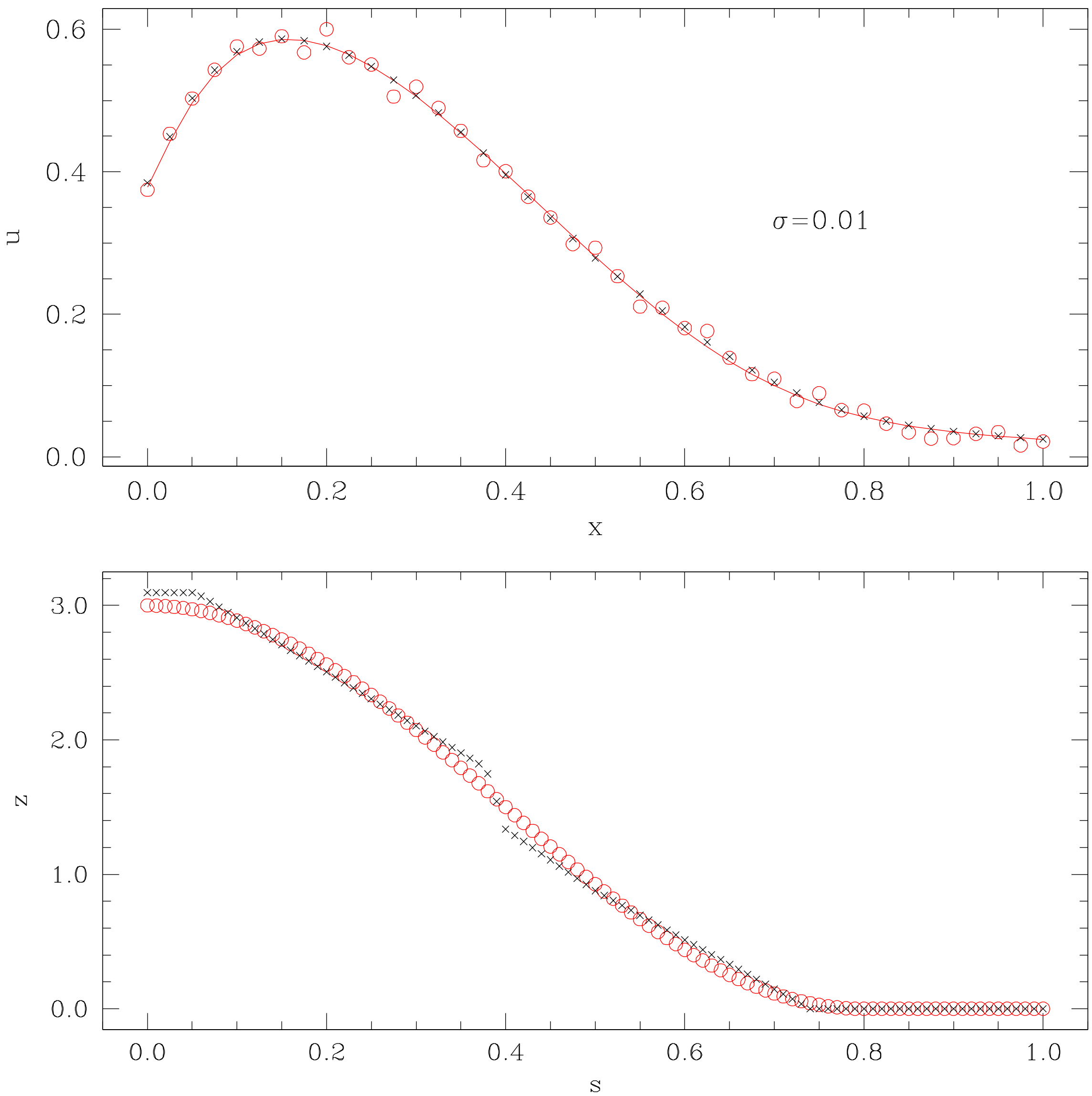

Figure 3: Solution 1.

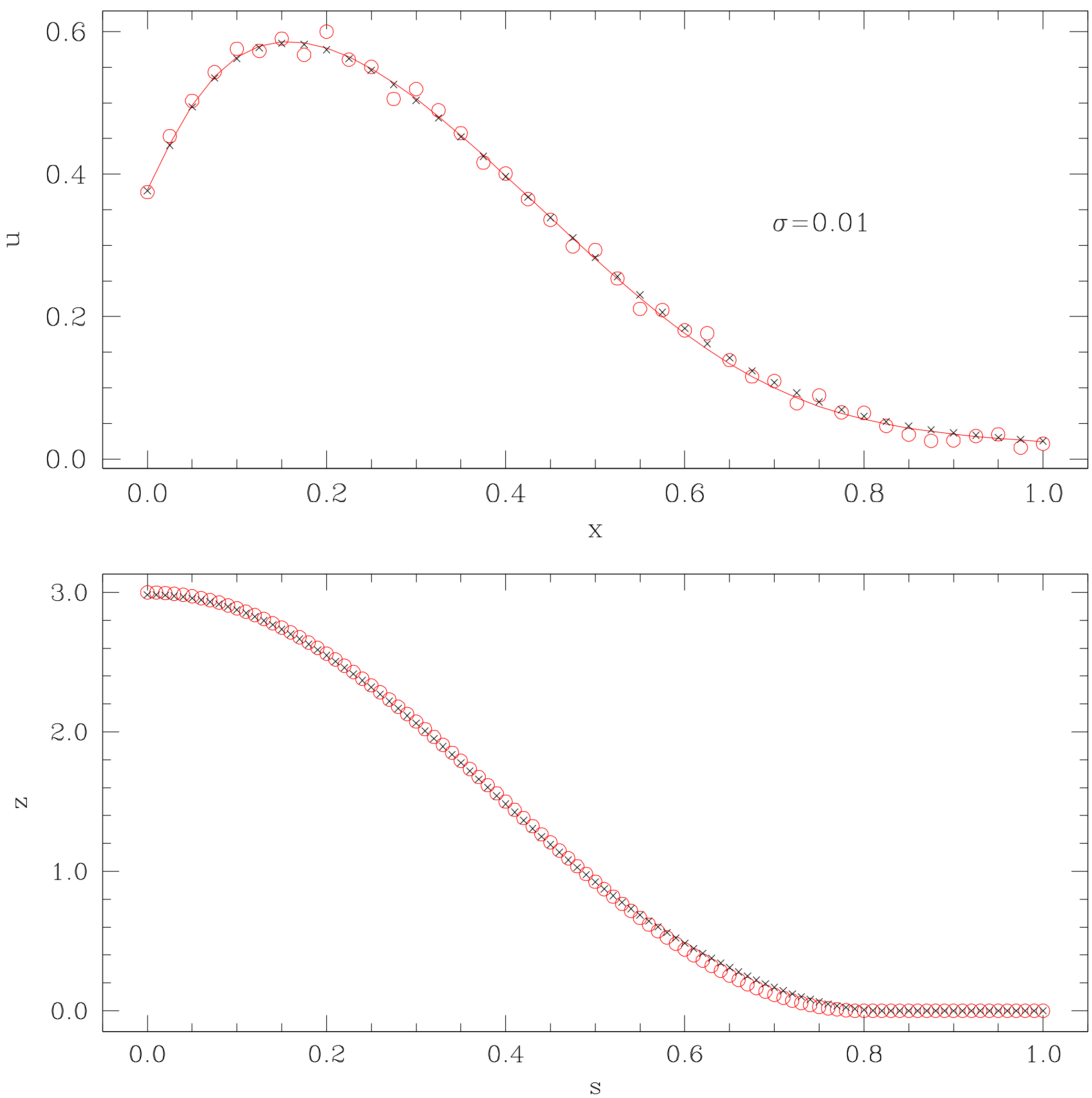

Figure 4: Solution 2.

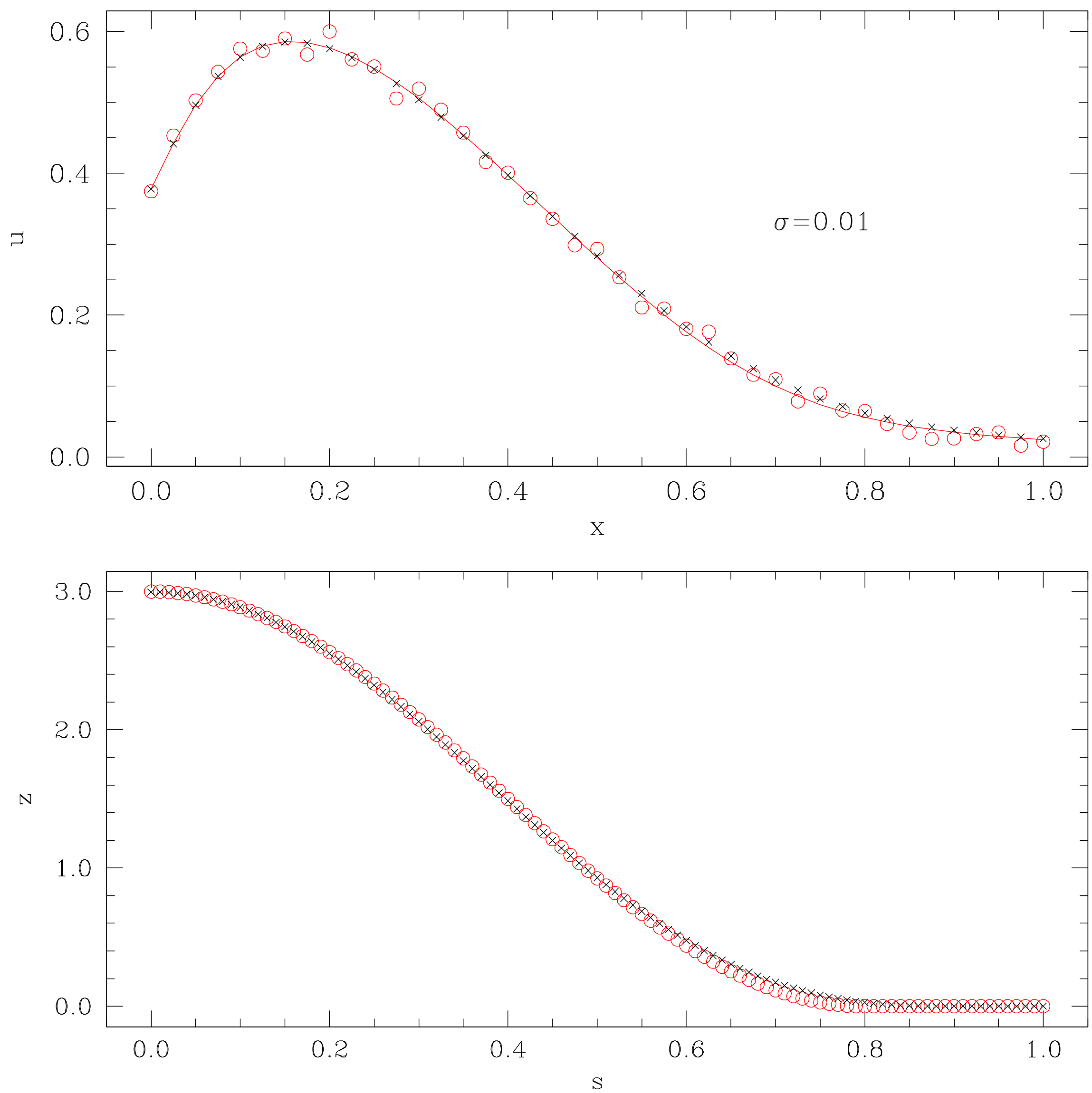

Figure 5: Solution 3.