SEPLib nonlinear solver library – Manual

Ali Almomin, Ettore Biondi, Yinbin Ma, Kaixi Ruan, Joseph Jennings, Robert Clapp, Musa Maharramov and Alejandro Cabrales-Vargas

ABSTRACT

We created an SEPLib nonlinear solver library that serves as the infrastructure for many inversion implementations. The library targets gradient-based inversion schemes with a focus on ease of use, flexibility, reusability and expandability. This was achieved by taking advantage of modern language features in object-oriented programing. This manual is divided in three parts. First, we detail what the library components are and explain all the abstract parts. Second, we show how to use each component of the library. Third, we show examples of different inversion implementations.

INTRODUCTION

The purpose of this manual is to provide a reference for the SEPLib nonlinear inversion library. The design of the library is based on a divide and conquer approach where we separate the inversion into four parts:

- **Problem definition** which includes the details of how the objective function and gradient are computed
- **Stepper** which finds a proper scalar to a given search direction to reduce the objective function
- **Terminator** which determines when to stop the inversion
- **Solver** which runs the iterative inversion by using the other three parts

Our goal is to make each part independant from the other parts. This separation increase reusability of any part that doesn't need to be changed between different inversion problems.

Each part has an abstract object that define its interface, variables and procedures. By starting with an abstract base, it is possible to create as many implementations as we need as long as they adhere to the abstract object requirements. This design allows for an easy expansion of the library with more implementations in the future. We also created several concrete implementations of each abstract object using conventional

inversion techniques. Moreover, we created a driver, i.e. a wrapper, that uses all the concrete implementations of the library.

The manual is divided in three sections. The first section details the library components and explains all the abstract objects for the users interested in the computer science part of the library. The second section shows how to create a concrete implementation of each abstract object and lists all implementation we created so far. This section targets users that are interested in creating their own solver and steppers or want to define their objective function with a nonconventional norm or residual. The third sections shows examples of different inversion problems targetting users that want to run the existing implementations.

ABSTRACT COMPONENTS

problem

The problem abstract objects mainly store functions pointers that calculate objective function, the residual, the gradient and the change in residual. However, the library separates different components of the inversion which can potentionally cause some recalculations to occur. Therefore, we decided to prevent such recalculation by following an encapsulation approach with this abstract object such that it stores a private local copy of the input and output variables with "get" procedures that output the needed variable. We added a simple dependency between the variable with "updated" logical flags such that changing the dependencies causes the variables to be recalculated. We decided to add output procedure that writes the local variables to disk as the inversion is running. Finally, we added counters to keep track of the number of evaluations and writes that occured so far since this is the object where the calculations occur.

The following shows the definition of the abstract object:

```
type, abstract :: prob_obj
  private
  integer, public :: nmodl = 0, ndata = 0
  integer :: fevals = 0, counter = 0
  logical :: obj_updated = .false., res_updated = .false.
  logical :: grad_updated = .false., dres_updated = .false.
  double precision :: obj = 0.
  real, dimension(:), allocatable :: modl, res, grad, dmodl, dres
  character(256), public :: rnorm = '', mmov = '', gmov = '', rmov = ''
end type
```

The variables `nmodl` and `ndata` store the size of the model and data space. The variables `fevals` and `counter` keep track of the number of evaluations and writes,

the variables ending with _updated are the update flag for each local variable. The variable modl is the model, res is the residual which depends on modl, obj is the objective function value which depends on both modl and res, grad is the gradient which depends on modl and res, dmodl is the model update and dres is the change in residual which depends on modl and dmodl. The variables rnorm, mmov, gmov and rmov are the output tags for the residual norm, model movie, gradient movie and residual movie, respectively.

The procedures of this abstract object are:

```
procedure :: objf, resf, gradf, dresf
procedure :: alloc, set_modl, output
procedure :: get_obj, get_res, get_grad, get_dres
procedure :: get_fevals, get_counter, get_rnorm, get_gnorm
```

The procedures objf, resf, gradf and dresf will store the user functions that calculate each respective variable. Notice that we did use deferred procedures although we need the user to impelment these functions. The reason will be explained later. The alloc procedure checks that the sizes were provided then allocates all the local variables as follows:

```
integer function alloc(this)
  class(prob_obj) :: this
  if(this%nmodl < 1 .or. this%ndata < 1) then
    call erexit("problem: nmodl or ndata is not correct")
  end if
  allocate(this%modl(this%nmodl))
  allocate(this%res(this%ndata))
  allocate(this%grad(this%nmodl))
  allocate(this%dres(this%ndata))
  allocate(this%dmodl(this%nmodl))
  alloc = 0
end function
```

The set_modl procedure, which is intended to be used internally in this object, checks whether the provided model is different than the local model and updates the local copy if needed. Also, this procedures changes the update flags accordingly as follows:

```
integer function set_modl(this, modl)
  class(prob_obj) :: this
  real, dimension(:) :: modl
  if(any(this%modl .ne. modl)) then
```

```
    this%modl = modl
    this%obj_updated = .false.
    this%res_updated = .false.
    this%grad_updated = .false.
    this%dres_updated = .false.
  end if
  set_modl = 0
end function
```

All the `get_` procedures do three steps. First, check if the size of the input is consistent with the local sizes. Then, use the procedure `set_modl` to update the local model (if needed), and finally, calculate the requested variable if it is not updated and add the number of evaluations to `fevals`. If it is updated, the procedure will simply output the local variable. For instance, the following code shows the `set_grad` procedure:

```
integer function get_grad(this, modl, grad)
  class(prob_obj) :: this
  real, dimension(:) :: modl, grad
  integer :: stat
  if(size(modl) .ne. this%nmodl .or. size(grad) .ne. this%nmodl) then
    call erexit("problem: size of modl or res is not correct")
  end if
  stat = this%set_modl(modl)
  if(.not. this%grad_updated) then
    stat = this%get_res(this%modl, this%res)
    stat = this%gradf(this%modl, this%res, this%grad)
    this%fevals = this%fevals + 2
    this%grad_updated = .true.
  end if
  grad = this%grad
  get_grad = 0
end function
```

The `output` procedure outputs the variables if their corresponding tag is used while using `counter` to keep track of how many times it wrote to disk. When writing for the first time, it also creates the history file for the output. The code of the procedure is:

```
integer function output(this, modl)
  class(prob_obj) :: this
  real, dimension(:) :: modl
```

```
  integer :: stat
  this%counter = this%counter + 1
  if(this%counter .eq. 1) then
    if(this%mmov .ne. '') then
      call to_history("n1",this%nmodl,this%mmov)
      call to_history("label1","Model index",this%mmov)
      call to_history("label2","iteration",this%mmov)
    end if
  end if
  if(this%mmov .ne. '') then
    call to_history("n2",this%counter,this%mmov)
    call srite(this%mmov, modl, 4*size(modl))
  end if
  output = 0
end function
```

The procedure `objf`, and similarly `resf`, `gradf` and `dresf`, is impelmented as follows:

```
integer function objf(this, res, obj)
  class(prob_obj) :: this
  real, dimension(:) :: res
  double precision :: obj
  call erexit("ERROR: objf was not overridden")
  objf = 0
end function
```

This implementation can be viewed as an optional deferred such that the user does not have to implement functions, such as `dresf`, unless needed by inversion. For the needed functions, the user should overwrite the procedure. While this approach is flexible, it has the downside that the errors will happen in run-time instead of compile-time.

stepper

The stepper abstract object has been designed in such a manner to allow for a variety of steppers to be implemented. In addition to this flexibility, this design also requires very little knowledge from the user when using any one of the steppers. In general, to describe our abstract stepper, we have decided to allow the user to clip the model by specifiying a model maximum (`mmax`) and minimum (`mmin`) in the procedure `clip_modl`. We also included a deferred `run` procedure that should do the actual execution of the stepper. The implementation of this design is shown in the following code:

```
type, abstract :: stepper_obj
    real, dimension(:), pointer :: mmin => null(), mmax => null()
  contains
    procedure(stepper_function), deferred :: run
    procedure :: clip_modl
end type
```

The abstract interface for the run routine is shown in the following code:

```
  abstract interface
    integer function stepper_function(this, modl, prob, dmodl, &
                                      alpha, success)
      import :: stepper_obj, prob_obj
      class(stepper_obj) :: this
      class(prob_obj) :: prob
      real,dimension(:) :: dmodl, modl
      double precision :: alpha
      logical :: success
    end function
  end interface
```

The abstract interface for the run routine requires the user to provide the model array `modl`, a problem object explained before `prob`, the search direction `dmodl`, an `alpha` variable to return the computed step size, and lastly a logical `success` variable used by the solver to check if the stepper has sucessfully computed a proper step size. If it does not find a proper step size, then when the solver finds that `success` is `.false.`, it will terminate the program.

The `clip_modl` procedure has a simple implementation such that the bounds are optional, as shown in the following function:

```
  integer function clip_modl(this, modl)
    class(stepper_obj) :: this
    real, dimension(:) :: modl
    if(associated(this%mmin)) where(modl < this%mmin) modl = this%mmin
    if(associated(this%mmax)) where(modl > this%mmax) modl = this%mmax
    clip_modl = 0
  end function
```

terminator

The terminator abstract object is composed of a deferred `test` function and it is defined as follows:

```
type, abstract ::  terminate_obj
contains
  procedure(terminate_test), deferred :: test
end type
```

where abstract interface of this function is written as:

```
abstract interface
  logical function terminate_test(this, iter, prob)
    import :: terminate_obj
    import :: prob_obj
    class(terminate_obj) :: this
    class(prob_obj) :: prob
  end function
end interface
```

The test procedure requires the iteration number `iter` and a problem object `prob`. The problem object is required because some stopping criteria depend on the residuals or the gradient of the problem.

solver

The solver abstract object is very simple. In fact, this object is composed only by a deferred `run` function and it is defined as follows:

```
type, abstract :: solver_obj
  contains
  procedure(run_solver),deferred :: run
end type
```

where abstract interface of this function is written as:

```
abstract interface
   integer function run_solver(this, modl, prob, stpr, term)
     import :: solver_obj, stepper_obj, terminate_obj
     import :: prob_obj
     class(solver_obj) :: this
```

```
      real, dimension(:) :: modl
      class(prob_obj) :: prob
      class(stepper_obj) :: stpr
      class(terminate_obj) :: term
    end function
  end interface
```

and requires as input the model array `modl`, a problem object explained before `prob`, a stepper object `stpr`, and a terminator object `term`. We decided to construct the abstract solver object without any initial variable to give more flexibility to a user that wants to construct its own solver. In fact, by including a variable in the abstract object, the user must declare this component in an actual solver object. It is also worth noticing that we separated the solver from the stepper. In this manner we can use different stepper algorithms in combination with an actual solver.

IMPLEMENTATIONS

problem

The problem objects contains the details of how to calculate the residuals, objective fucntion, gradient, and changes in the gradient. When creating a concrete implementation of this object, the user needs to do the following steps:

1. extend the abstract object
2. set the values of the variables `ndata` and `nmodl`
3. run the `alloc` procedure
4. overwite the procedures `objf`, `resf`, `gradf` and `dresf`

The first two steps can be done with a user-defined constructor and the last step need to be included as type-bound procedures. As long as these steps are satisfied, the user is free to add modify any number of variables or procedures for the inversion problem.

We created a concrete implmentation that minimizes a nonlinear objective function of the form:

$$J(\mathbf{m}) = \frac{1}{2}\|\mathbf{W_d}\left(\mathbf{F}(\mathbf{S}\mathbf{W_m}\mathbf{m}) - \mathbf{d}_{\text{obs}}\right)\|_2^2, \tag{1}$$

where $\mathbf{W_d}$ is a data weighting operator, $\mathbf{F}$ is a nonlinear forward operator, $\mathbf{S}$ is a preconditioning operator, $\mathbf{W_m}$ is a model weighting operator, $\mathbf{m}$ is the model and $\mathbf{d}_{\text{obs}}$ is the observed data. We made all the operators, except the foward modeling operator, optional by initializing them to an identity operator. For this problem, the object definition is:

```
type, extends(prob_obj) :: prob_nl_l2_diff
  private
  integer :: nsop
  real, dimension(:), pointer :: data
  real, dimension(:), allocatable :: temp_data, temp_modl, temp_prec
  procedure(oper), nopass, pointer :: fop, lop, wdop, wmop, sop
  procedure(update), nopass, pointer  :: uop
contains
  private
  procedure   :: objf, resf, gradf, dresf
end type
```

where `nsop` is the size of the preconditioned variable, `data` is the observed data, `temp_` are temporary variables, and the `op` pointers are procedure pointers to the operators. Notice that we added two operators: `lop` to contain the linearized operator required to calculate the gradient and `uop` to contain the operator that updates the model for the operators. We then defined an interface block to create a user-defined constructor and specify the interface of all the operators as:

```
interface prob_nl_l2_diff
  module procedure const
end interface
interface
  integer function oper(adj, add, modl, data)
    real,dimension(:) :: modl,data
    logical :: adj,add
  end function
  integer function update(modl)
    real,dimension(:) :: modl
  end function
end interface
```

where the interfaces follow the conventional SEPLib operators. The user-defined constructor satisfy the requirements we mentioned earlier as follow:

```
type(prob_nl_l2_diff) function const(fop, uop, lop, nmodl, ndata, data, &
                                     wdop, wmop, sop, nsop)
  optional :: wdop, wmop, sop, nsop
  procedure(oper) :: fop, lop, wdop, wmop, sop
  procedure(update) :: uop
  integer :: nmodl, ndata, nsop, stat
  real, dimension(:), target :: data
  const%fop => fop
```

```
  const%uop => uop
  const%lop => lop
  const%nmodl = nmodl
  const%ndata = ndata
  const%data => data
  const%wdop => identity_op
  const%wmop => identity_op
  const%sop => identity_op
  const%nsop = nmodl
  if(present(wdop)) const%wdop => wdop
  if(present(wmop)) const%wmop => wmop
  if(present(sop)) const%sop => sop
  if(present(nsop)) const%nsop = nsop
  allocate(const%temp_data(const%ndata))
  allocate(const%temp_modl(const%nmodl))
  allocate(const%temp_prec(const%nsop))
  stat = const%alloc()
end function
```

The only thing left is to calculate each variable using its function. The objective function is calculated using the residual in `objf` is:

```
integer function objf(this, res, obj)
  class(prob_nl_l2_diff) :: this
  real, dimension(:) :: res
  double precision :: obj
  obj = 0.5*(sum(dprod(res,res)))
  objf = 0
end function
```

The residual is calculate using the model in the `resf` function as:

```
integer function resf(this, modl, res)
  class(prob_nl_l2_diff) :: this
  real, dimension(:) :: modl, res
  integer :: stat
  stat = this%wmop(F, F, modl, this%temp_modl)
  stat = this%sop(F, F, this%temp_modl, this%temp_prec)
  stat = this%fop(F, F, this%temp_prec, this%temp_data)
  this%temp_data = this%temp_data - this%data
  stat = this%wdop(F, F, this%temp_data, res)
  resf = 0
end function
```

The gradient is calculated using the model and residual in the `gradf` function as:

```
integer function gradf(this, modl, res, grad)
  class(prob_nl_l2_diff) :: this
  real, dimension(:) :: modl, res, grad
  integer :: stat
  stat = this%uop(modl)
  stat = this%wdop(T, F, this%temp_data, res)
  stat = this%lop(T, F, this%temp_prec, this%temp_data)
  stat = this%sop(T, F, this%temp_modl, this%temp_prec)
  stat = this%wmop(T, F, grad, this%temp_modl)
  gradf = 0
end function
```

The change in residual is calculate using the model and model update in the `dresf` function as:

```
integer function dresf(this, modl, dmodl, dres)
  class(prob_nl_l2_diff) :: this
  real, dimension(:) :: modl, dmodl, dres
  integer :: stat
  stat = this%uop(modl)
  stat = this%wmop(F, F, dmodl, this%temp_modl)
  stat = this%sop(F, F, this%temp_modl, this%temp_prec)
  stat = this%lop(F, F, this%temp_prec, this%temp_data)
  stat = this%wdop(F, F, this%temp_data, dres)
  dresf = 0
end function
```

We also created an implementation for linear problems and another implementation that combines two problems by summing them.

stepper

The abstract stepper class has been extended to several more specific stepper classes, such as the backtracking or parabolic steppers. In general, a stepper class derived from the abstract class is done as follows:

```
type, extends(stepper_obj) :: mystepper
contains
  procedure :: run
end type
```

Regardless of the stepper, it must override the run procedure that it inherits from the abstract class. This is done by first declaring the `run` function in the derived type as seen above, and later within the function definition the user writes the actual stepper algorithm. One example of this can be shown with the parabolic stepper class. In the following code listing, we have derived the `stepper_parab` type (specific stepper) from the `stepper_obj` type (abstract stepper) we omit the definition of `run` for brevity:

```
type, extends(stepper_obj) :: stepper_parab
  integer :: c1 = 1.0, c2 = 2.0
  double precision :: ntry = 10, r = 0.25, alpha = 0.
  double precision :: alpha_min = 0.25, alpha_max = 4.0
  real, dimension(:), allocatable :: modl, res, dres
contains
  procedure :: run
end type
```

If the user then desires to implement their own stepper they can do so with a simple call (again using the parabolic stepper as the example):

```
type(stepper_parab) :: parab1
```

and can make changes to the members of the `parab1` object via the following calls:

```
parab1%ntry=1000
parab1%c1=-200.0
parab1%c2=200.0
```

We create one concrete line search class based on backtracking method. The method looks for a model update that satisfies Wolfe condition, as follows:

$$J(m_i + \alpha \Delta m_i) \leq J(m_i) + c_1 \alpha \mathbf{g_i}^T \nabla J(m_i) \tag{2}$$

$$|\mathbf{g_i}^T \nabla J(m_i) + \alpha \Delta m_i| \leq c_2 \mathbf{g_i}^T \nabla J(m_i), \tag{3}$$

where c_1 and c_2 need to be set. The backtracking method start from $\alpha = \alpha_{max}$ and then $\alpha \times \rho \rightarrow \alpha$ at each step until the Wolfe condition is satisfied or we reached minmum allowed α_{min}.

Therefore we extend the line search method as:

```
type, extends(stepper_obj) :: stepper_backtrack
  double precision :: decrease_coef = 0.01, curvatur_coef = 0.5
  double precision :: alpha = 0., alpha_min = 0.25, alpha_max = 4
  double precision :: rho = 0.5
  real, dimension(:), allocatable :: modl, res, dres, grad
contains
  procedure :: run
end type
```

where c_1 (**decrease_coef**), c_2 (**curvatur_coef**), ρ, α_{min} and α_{max} are specific to back tracking line search method. Those parameters has default value and can also be set by the experienced users.

We also created a few more line search implementations, such as the a sampling line search and a parabolic interpolation which is Brent's method. For Brent's method we have input a and b and try to find $a \leq \alpha b$ that minimize the objective function. The algorithm keep shrinking the interval $[a, b]$ until $J(m + a\Delta m) - J(m + b\Delta m)$ or $|a - b|$ is below a certain threshold.

terminator

The terminator object determines when to stop the inversion. This is done by the procedure `test` which returns a logical value on whether the inversion should be terminated. We created a concrete implementation that gives the user the option to stop using one of the following criteria: number of iterations, number of evaluations, number of hours passed, size of residual norm and size of gradient norm. The object definition is:

```
type, extends(terminate_obj) :: terminate_simple
  integer :: niter = 0, maxfevals = 0
  real :: maxhours = 0.
  double precision :: tolr = 1.0d-20
  double precision :: tolg = 1.0d-20
  integer(kind=8), private :: timer0 = 0, rate = 0
contains
  procedure :: test
end type
```

Because we need to keep track of time, we created a user-defined constructor that initializes the time passed as follows:

```
interface terminate_simple
  module procedure const
```

```
  end interface

  type(terminate_simple) function const(niter, maxfevals, maxhours, tolr, tolg)
    optional :: niter, maxfevals, maxhours, tolr, tolg
    integer :: niter
    integer :: maxfevals
    real :: maxhours
    double precision :: tolr
    double precision :: tolg
    if(present(niter)) const%niter = niter
    if(present(maxfevals)) const%maxfevals = maxfevals
    if(present(maxhours)) const%maxhours = maxhours
    if(present(tolr)) const%tolr = tolr
    if(present(tolg)) const%tolg = tolg
    call system_clock(const%timer0, const%rate)
    const%rate = const%rate * 3600.
  end function
```

The implementation of the `test` procedure checks if the user provided any stopping criteria and then test for that criteria accordingly as follows:

```
  logical function test(this, iter, prob)
    class(terminate_simple) :: this
    integer :: iter
    class(prob_obj) :: prob
    real :: hours_passed
    integer(kind=8) :: timer1
    test = .false.
    if(this%niter > 0 .and. iter > this%niter) then
      test = .true.
      write(0,*) "Terminate: maximum number of iterations reached"
    end if
    if(this%maxfevals > 0 .and. prob%get_fevals() >= this%maxfevals) then
      test = .true.
      write(0,*) "Terminate: maximum number of evaluations"
    end if
    if(this%maxhours > 0.) then
      call system_clock(timer1)
      hours_passed = 1.*(timer1-this%timer0)/this%rate
      if(hours_passed >= this%maxhours) then
        test = .true.
        write(0,*) "Terminate: maximum number hours reached", &
                   hours_passed, this%maxhours
      end if
```

```
    end if
    if(prob%get_rnorm() < this%tolr) then
      test = .true.
      write(0,*) "Terminate: minimum residual tolerance reached", &
                 real(prob%get_rnorm()), real(this%tolr)
    end if
    if(prob%get_gnorm() < this%tolg) then
      test = .true.
      write(0,*) "Terminate: minimum gradient tolerance reached", &
                 real(prob%get_gnorm()), real(this%tolg)
    end if
  end function
```

solver

When implementiong the solver object, the user needs to do the following in a loop:

1. run the problem object procedure `output`
2. calculate the search direction
3. run the stepper by the procedure `run` and test its success
4. check the terminator procedure `test` to determine when to stop the loop

If the solver doesn't require a line search, then the user need to implement the stepper procedure `clip_modl` every time the model is modified.

To show how a user can define an actual solver object, we describe the extension of the abstract solver to a non-linear conjugate-gradient solver object. First, we need to declare the extension of the abstract solver object as follows:

```
  type, extends(solver_obj) :: solver_nlcg
    character(20) :: beta_type='FR'
    contains
    procedure :: run
  end type
```

where the string `beta_type` defines how the step length is computed at a given iteration with the previous search direction and the current gradient. We see that in the children object `solver_nlcg` contains a `run` function as the abstract solver object. In this actual object we cannot define just a simple abstract interface. In fact, we need to implement the solver function keeping the same interface of the abstract interface (i.e., `integer function run(this, modl, prob, stpr, term)`). In this function we declare all the temporary variables that we need to run the solver. In our case this interface is written as follows:

```
class(solver_nlcg) :: this
integer :: stat
real, dimension(:) :: modl
class(prob_obj) :: prob
class(stepper_obj) :: stpr
class(terminate_obj) :: term
double precision :: beta, alpha, f
integer :: iter
logical :: success
real, dimension(:), allocatable :: grad, grad0, dmodl
```

where `alpha` is the step size, `beta` is the conjugate gradient ration between current and previous search directions, `f` is the objective function value, `grad` is the gradient, `grad0` is the previous gradient and `dmodl` is the search direction. The inversion loop is very simple and it is implemented as:

```
beta = 0.
iter = 1
do
  grad0 = grad
  stat = prob%get_grad(modl, grad)
  stat = prob%output(modl)
  if(iter > 1) beta = betaf(grad, grad0, dmodl)
  if(beta < 0.) beta = 0.
  dmodl = beta*dmodl - grad
  stat = stpr%run(modl, prob, dmodl, alpha, success)
  if(.not. success) then
    write(0,*) "Stepper couldn't find a proper step size"
    exit
  end if
  stat = prob%get_obj(modl, f)
  write(0,*) "iter", iter, " obj =", real(f), " feval =", prob%get_fevals()
  iter = iter + 1
  if(term%test(iter, prob)) exit
end do
```

First, we start the loop by placing the previous gradient into the `grad0` variable. Then, we retrieve or compute the current gradient calling the problem object's function `get_grad` and output the previous model calling the `prob%output` function. After these steps, we compute the conjugate search direction with the `betaf` function. At the first iteration the search direction is equal to the current gradient that means stepping with the steepest descent direction. Having computed the search direction we can now call the stepper and find the step length with the `stat = stpr%run(modl, prob, dmodl, alpha, success)` line. From the result of the stepper contained in

the `success` variable, we can decide whether a better model has been found or not. The last steps are to print some useful information to screen and test if the termination criteria has been achieved. The beta function is implemented separately from the solver `run` function (see the file `solver_nlcg.f90` in the `Src` directory). The reader interested in the mathematical details of non-linear conjugate gradient can find them in Dai and Yuan (1999).

We also created an implementation for the L-BFGS solver and the linear conjugate-gradient solver.

driver

We created a driver to simplify using a library, particularly for users that only want to use one of the concrete implementations in the library. This driver allows the user to either provide objects for each of the solver, stepper and terminator or just specify the number of each object. The driver has one function which has the following interface:

```
integer function nlsolver(modl, prob, nsolver, nstepper, niter, &
                          solver, stepper, term)
  optional :: nsolver, nstepper, niter, solver, stepper, term
```

The variable `modl` is the model, `prob` is the problem definition object, `nsolver` is the solver number, `nstepper` is the stepper number, `niter` is the number of iterations, `solver` is the solver object, `stepper` is the stepper object and `term` is the terminator object. If both the number and the object are provided, the driver will prioritize the object using a local pointer. If neither is provided, the driver will pick the default options which are the nonlinear conjugate-gradient solver and back-tracking stepper. The following is an example of picking the solver:

```
if(present(solver)) then
  solverl => solver
else
  nsolverl = NLCG
  if(present(nsolver)) nsolverl = nsolver
  if(nsolverl .eq. NLCG) then
    solver1 = solver_nlcg()
    solverl => solver1
  else if(nsolverl .eq. BFGS) then
    solver2 = solver_lbfgs()
    solverl => solver2
  else if(nsolverl .eq. LCG) then
    solver3 = solver_lcg()
    solverl => solver3
```

```
    end if
  end if
```

After setting the solver, stepper and terminator, the driver will run the solver as:

```
stat = solverl%run(modl, prob, stepperl, terml)
```

To execute the driver, all we need is to run the following line:

```
stat = nlsolver(modl, prob1, NLCG, PARAB, niter=10)
```

EXAMPLES

Example 1

The first example we demonstrate is a linear missing data problem from chapter three of Geophysical Image Estimation by Example (GEE). The problem boils down to the following: given the filter in Figure 2 and the data in Figure 1, how can we "ensure that the restored data, after specified filtering, has minimum energy"? Now with the problem in mind, we form the following fitting goal:

$$\mathbf{0} \approx \mathbf{FJm} + \mathbf{Fm}_{\text{known}}$$

where $\mathbf{F}$ is the transient convoultion operator, $\mathbf{J}$ is a masking operator, $\mathbf{m}$ is a vector that contains all of the points shown in Figure 1 and $\mathbf{m}_{\text{known}}$ is a vector containing only the known data points. For a full derivation of this fitting goal, please refer to GEE page 72.

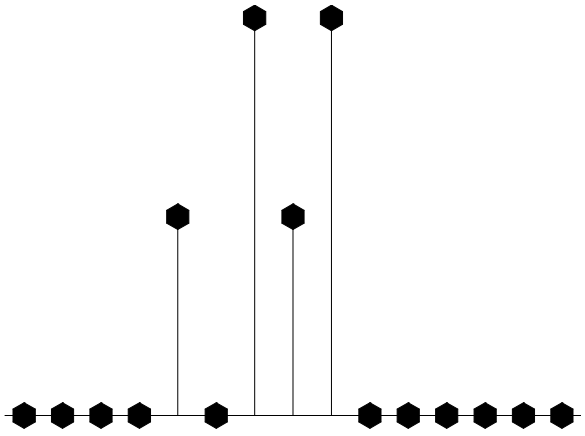


Figure 1: The data which we are attempting to restore. [**ER**]

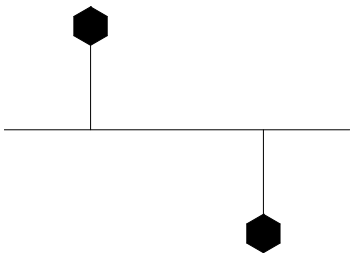


Figure 2: The filter which will be used to restore the data. [**ER**]

While this problem can be readily solved with a linear optimization scheme, for the purpose of this report, we solve this problem with a non-linear conjugate gradient solver and a parabolic fitting stepper. Also, we use the transient convolution and mask operators defined within GEELIB and supply these operators to our solver. To solve this problem with our linear solver, we will need two files. One of the files will be our main program that will do the I/O and call the solver as well as other necessary routines. The second file serve to define our problem object. In total, in order to solve this problem, the user would be required to write less than 30 lines of code. Here we go describe the code in detail:

Main program

Without the I/O and usual operator initialization, the call to perform the nonlinear optimization is shown in the following lines of code:

```
type(prob_datafill) :: prob
prob = prob_datafill(nmodl=nmodl,ndata=ndata,na=nain,data=datain)
stat = prob%alloc()
stat = nlsolver(modl, prob, NLCG, PARAB, niter=50)
```

In the first line, we create an instance of our problem object which we need to define in another module file (explained later). Then we can call our `prob` object's constructor to assign it the necessary variables (the arguments passed to the constructor). Then, with the call `prob%alloc()`, we allocate our model, residual, gradient, dmodel, and dresidual arrays that will be used in our non-linear conjugate gradient solver. With the last line, we call the non-linear solver passing it the `modl` array, the user-defined `prob` object, the type of solver, the type of stepper and finally the number of iterations. It is important to note that the only two required arguments for this `nlsolver` are the `modl` and `prob` arguments. The remainder can be left out and will be left out and will be handled by the solver. Also, while it was not done in this example, we can create stepper and solver objects that will allow us to to override default parameters set in the solvers and steppers.

In the second file, we must define the `prob_datafill` class as was shown previously in the problem section of this report. This means defining the previously mentioned `objfunc`, `resfunc`, `gradfunc`, and `dresfunc` functions, as well as a user-defined constructor that allows the user to pass additional values to the solver.

Once we have defined the main program as well as our `prob_datafill` class, we can implement our solver which prints the following at the command line:

```
Bin/Data_fill.x <  ./Dat/datamiss.H filtin=./Dat/mlines.H > fill_lines.H
 iter           1  obj =  0.6250000      feval =          11
 iter           2  obj =  0.5392157      feval =          16
 iter           3  obj =  0.4687500      feval =          21
 iter           4  obj =  0.4417293      feval =          26
 iter           5  obj =  0.4227941      feval =          31
 iter           6  obj =  0.4129246      feval =          36
 iter           7  obj =  0.4102488      feval =          41
 iter           8  obj =  0.4096886      feval =          46
 iter           9  obj =  0.4089912      feval =          51
 iter          10  obj =  0.4089340      feval =          56
 iter          11  obj =  0.4089286      feval =          64
 iter          12  obj =  0.4089285      feval =          72
 Stepper couldn't find a proper step size, will terminate solver
```

where `iter` is the iteration number, `obj` is the value of the objective function, and `feval` is the number of function evaluations.

In addition to diagnostic messages shown above, it gives us the result shown in Figure 3(a). In comparing this result to Jon's original result, we can observe that they are nearly equivalent.

Example 2

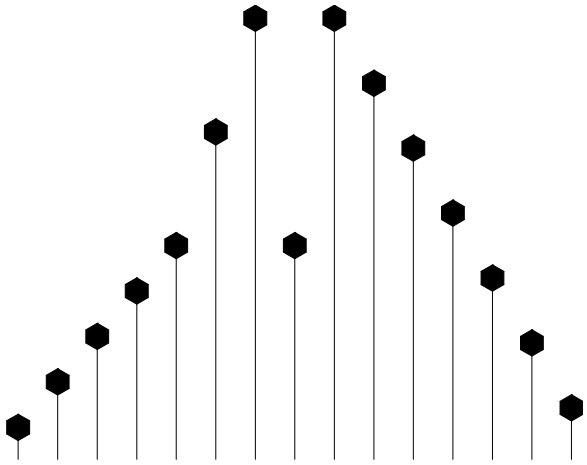
Rosenbrock function

To test our nonlinear solver we implement the well-known Rosenbrock function. The multidimensional Rosenbrock function is expressed as follows:

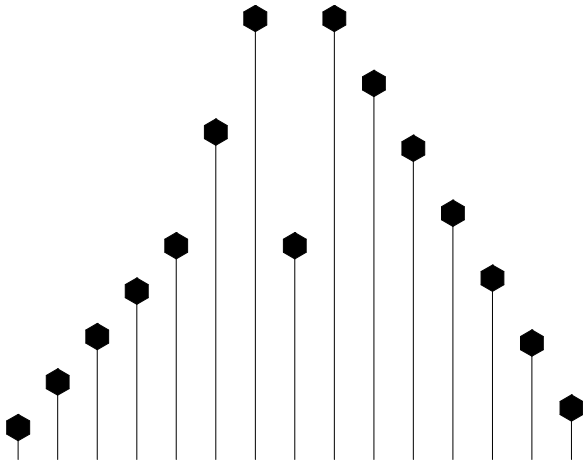
$$f(\mathbf{x}) = \sum_{i=1}^{N-1} \left[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2 \right], \tag{4}$$

where f is the objective function and $\mathbf{x}$ is the optimization parameter. In 2D this function reduces to:

$$f(x, y) = (1 - x)^2 + 100(y - x)^2, \tag{5}$$



(a)



(b)

Figure 3: (a) Missing data fill with the non-linear solver and (b) missing data fill from Claerbout's GEE. [**ER**]

and is shown in Figure 4. The 2D Rosenbrock function is know to have a single global minimum at $x = 1$ and $y = 1$ inside a broad minimum valley. When we start the optimization from $x = -1$ and $y = -1$ our nonlinear conjugate gradient algorithm converges in 92 iteration to $x = 1.000112$ and $y = 1.000224$. The optimization path is displayed in Figure 4 by the cyan line and the tested points are indicated by the purple stars. As shown by this path the algorithm has found the correct minimum and stopped close it. The source code for running the example is in the `Src` directory and named `Rosenbrock_test.f90`. This program handles multidimensional Rosenbrock functions.

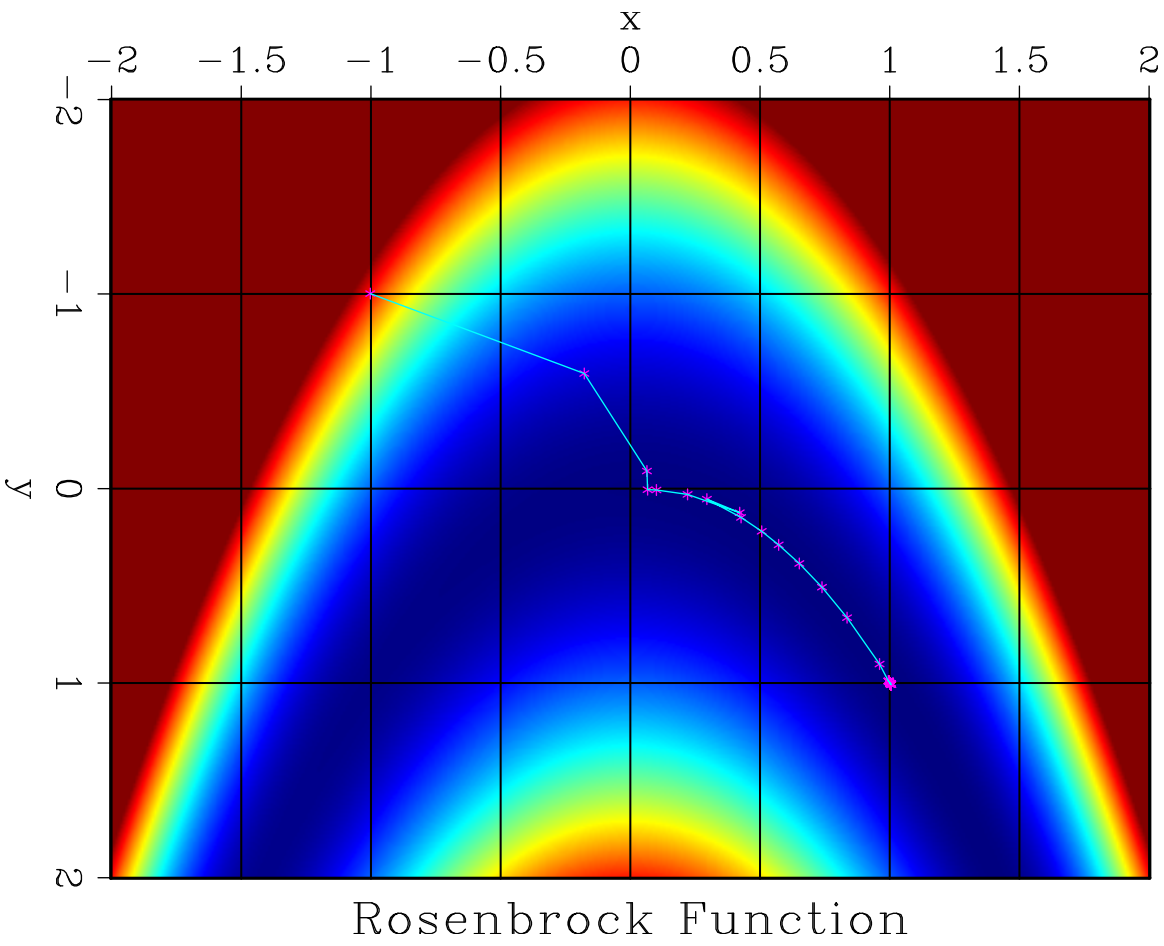


Figure 4: Bidimensional Rosenbrock function. Red and blue colors indicate high and low function values respectively. The cyan line with the purple stars is the optimization path followed when the starting solution is the point $x = -1$ and $y = -1$. [**ER**]

Example 3

Deghosting

Deghosting, together with debubbling, is one of the fundamental preprocessing techniques necessary for dealing with marine datasets. If we know the filter that generates the ghost frequency notch in the data is easy to suppress its effect. One of the biggest challenge is actually to estimate the filter coefficient for the ghost. The ghost effect for a single trace and vertical propagation can be expressed in the frequency domain as:

$$d(\omega) = m(\omega)(1 - Re^{\omega\tau}), \tag{6}$$

where $d(\omega)$ represents the ghosted trace, $m(\omega)$ is unghosted data, R is the reflection coefficient at the water surface, and τ is the two-way traveltime from the source to the water surface. Knowing the R and τ in this simple 1D case is easy to eliminate the ghost effect by deconvolving $d(\omega)$ with the filter $(1 - Re^{\omega\tau})$. However, estimating these two parameters is not a simple task in the real work. In fact, the ghost, in this case the source ghost, can be hidden by the bubble, and other physical phenomena. In addition, the water surface reflection coefficient can be frequency dependent. In this simple example where are assuming that R is independent of the frequency.

In order to estimate the two physical parameters of the ghost filter we need to decide an objective function to minimize. A good target to reach is minimizing the energy of the deghosted trace, since the unghosted data should contain less energy than the ghosted one. This objective function can be written as follows:

$$E(R, \tau) = \int \frac{d^*d(\omega)}{(1 - 2cos(\omega\tau) + R^2)} d\omega, \tag{7}$$

where E is the energy of the deghosted data, and $d^*d(\omega)$ is the amplitude spectrum of the ghosted trace. Knowing the analytical expression of the objective function (equation 7) is trivial to compute the partial derivative with respect each parameter and perform a gradient-based nonlinear inversion. We test the nonlinear solver library on two synthetic cases and one real case.

The left panel of Figure 5 shows a synthetic Gaussian impulse with $15Hz$ as central frequency, we consider this wavelet our unghosted trace. The right panel of the same figure displays this wavelet after applying a ghosting filter with $R = 0.6$ and $\tau = 50ms$. Since we know analytical expression of the objective function, we can compute it for certain range of values of the two parameters. Figure 7 shows the objective function computed for different value of the reflection coefficient and ghosting lag, it exhibits a clear global minimum, particularly elongated along the R-axis. The shape of the global minimum reflects the well-known fact that the energy of deghosted data is less sensitive to the water surface reflection coefficient. This observation is also shown by the superimposed optimization path that does not change in R as we progress toward the minimum when we start with $R = 0.5$ and $\tau = 70ms$. On the left panel of Figure

7 is shown a section of the energy fixing τ to the correct value. We see that close to the true reflection coefficient the gradient of the function is close to zero. In fact, the solver does not change the R model from the starting point. On the other hand, since the gradient is higher for the τ parameter, the solver can find a solution close to the correct value. In the right panel of Figure 7 is displayed the deghosted data using inverted parameter ($R = 0.499$ and $\tau = 51.845ms$). Most of the ghost energy has been canceled; however, since the R value is not correct we are introducing undesired ringing effects.

In the second synthetic case we use a different wavelet than before (left panel on Figure 8). In this case the wavelet is composed of two impulses of the same amplitude. We apply the ghost filter used in the previous case and we obtain the trace shown in the right panel of Figure 8. In this case we have obtain a wavelet similar to a Ricker. We can again compute the objective function for different values of the filter (Figure 9). In this case we notice that we have a more complex valley. This shape is caused by the presence of the second impulse in the original wavelet that is considered as a ghost effect. In this case if we start the solver from $R = 0.4$ $\tau = 55ms$ optimizes both model parameters. For this example because the of the presence of the second impulse, the minimum is centered on higher value of R respect to the correct solution (left panel of Figure 10). Deconvolving the ghosted trace with the optimized filter parameter, we eliminate most of the ghost energy but we obtain ringing effects as before.

The last test is performed on a real trace recorded in the Seneka lake from airgun testing (Figure 11). Looking at the energy plot as function of R and τ (Figure 12), we can see multiple minima for short delay time. These minima are cause by the presence of the bubble effect in the trace. However, we can easily estimate the correct delay thanks to presence of a precursor and its ghost at approximately $60ms$ and $92ms$ respectively. In fact, if we start from a starting parameters equal to $R = 0.4$ and $tau = 33ms$ the solver find the minimum of the valley where we start the optimization. However, the presence of the bubble influences this minimum and does not allow us to find the optimal deghosting filter (see right panel of Figure 13).

ACKNOWLEDGEMENT

The authors would like to thank Dolphin Geophysical and Chelminski associates for the permission to show the lake data.

REFERENCES

Dai, Y.-H. and Y. Yuan, 1999, A nonlinear conjugate gradient method with a strong global convergence property: SIAM Journal on Optimization, **10**, 177–182.

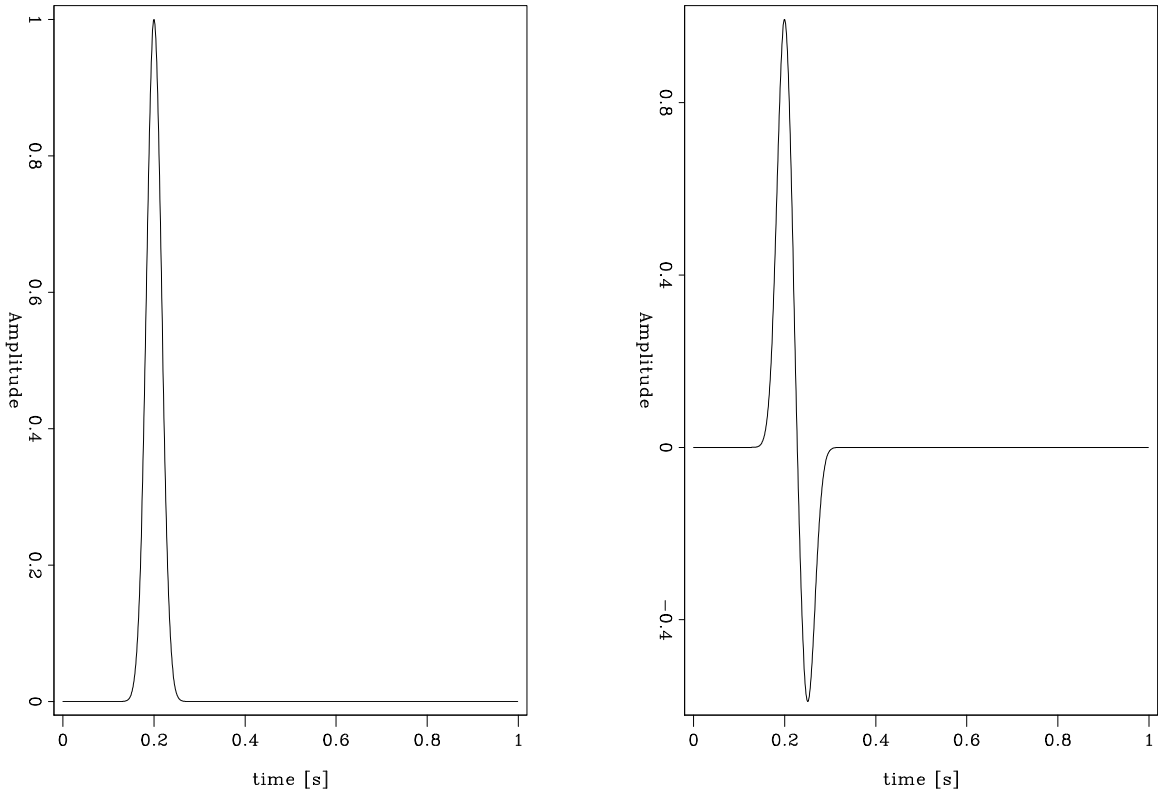


Figure 5: (left) Original unghosted trace. (right) Ghosted trace with ghosting parameters equal to $R = 0.6$ and $\tau = 50ms$. [**ER**]

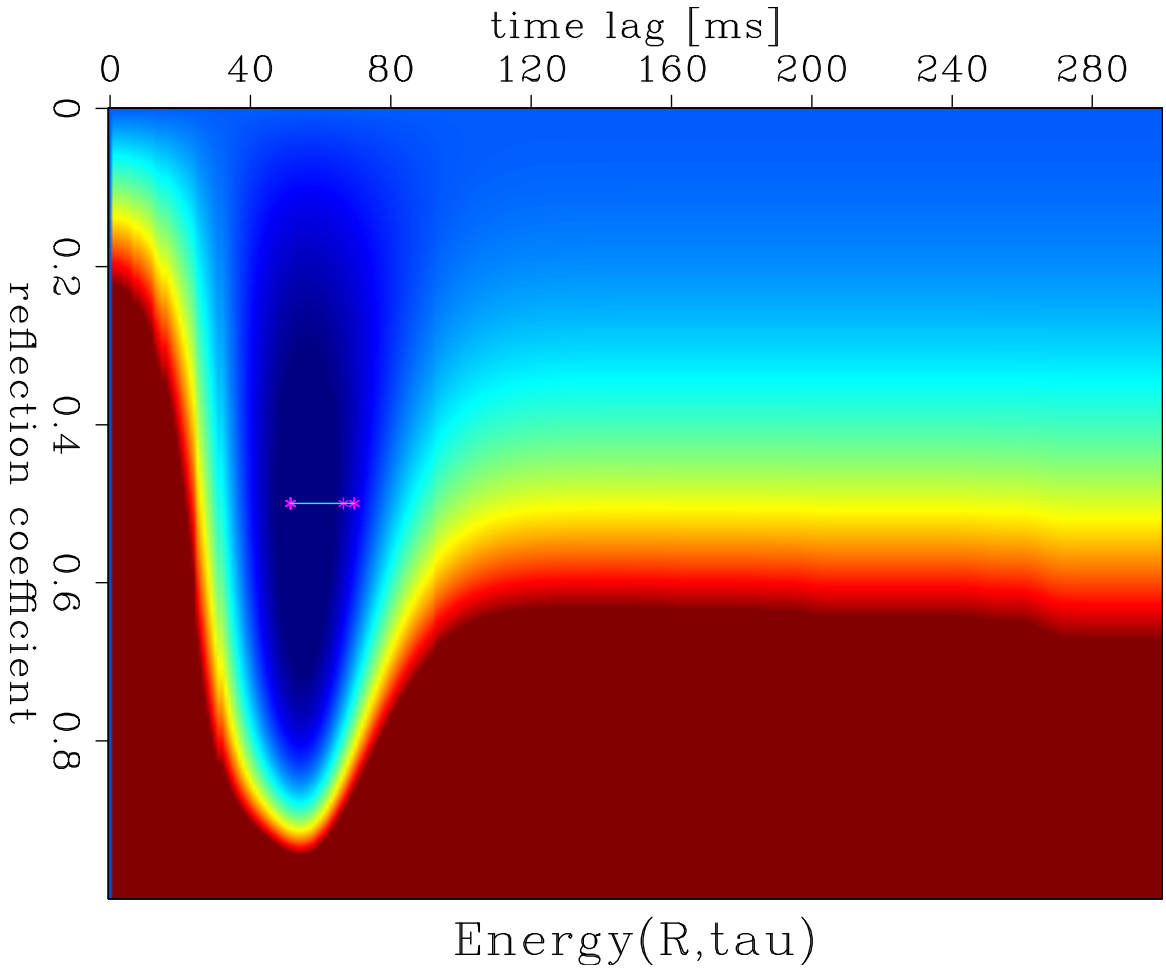


Figure 6: Energy of the ghosted trace of right panel Figure 5 as function of R and τ. Overlaid is the optimization path when the starting solution is $R = 0.5$ and $\tau = 70ms$. [**ER**]

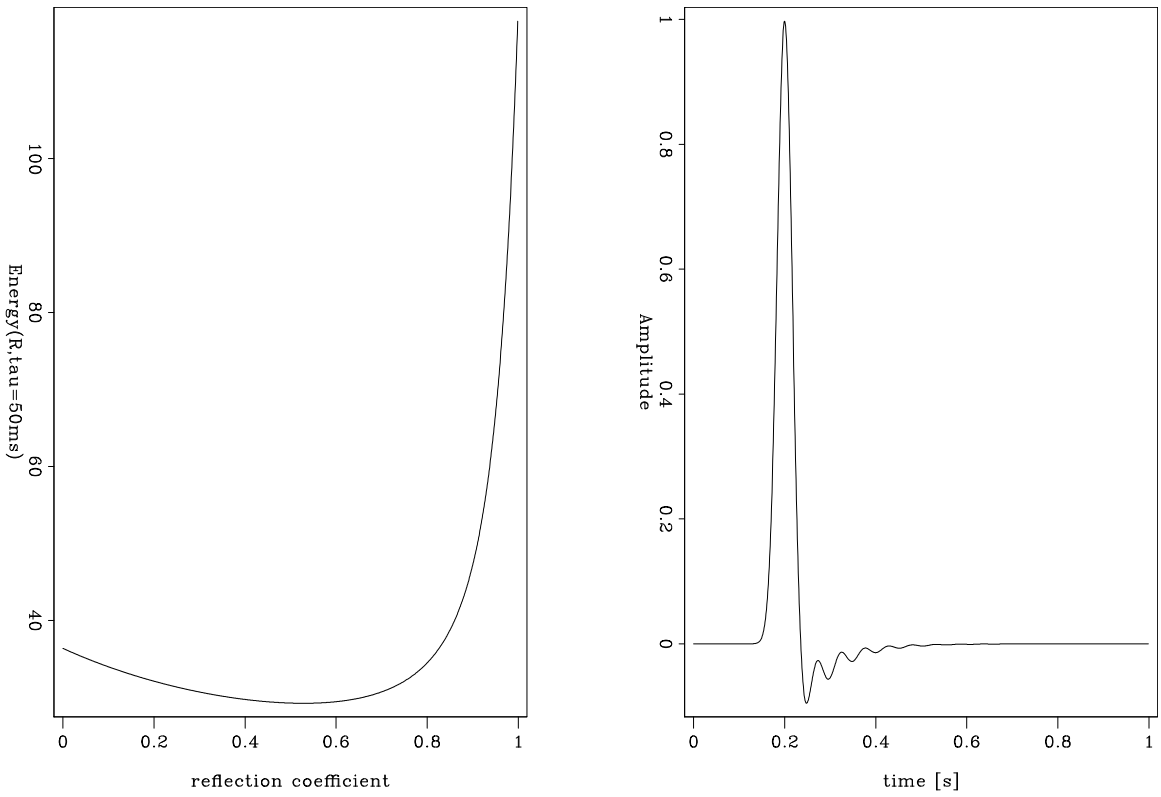


Figure 7: (left) Section of the objective function of left panel of Figure 6 for $\tau = 50ms$. (right) Deghosted trace using the optimized parameters from the solver ($R = 0.499$ and $\tau = 51.845ms$). [**ER**]

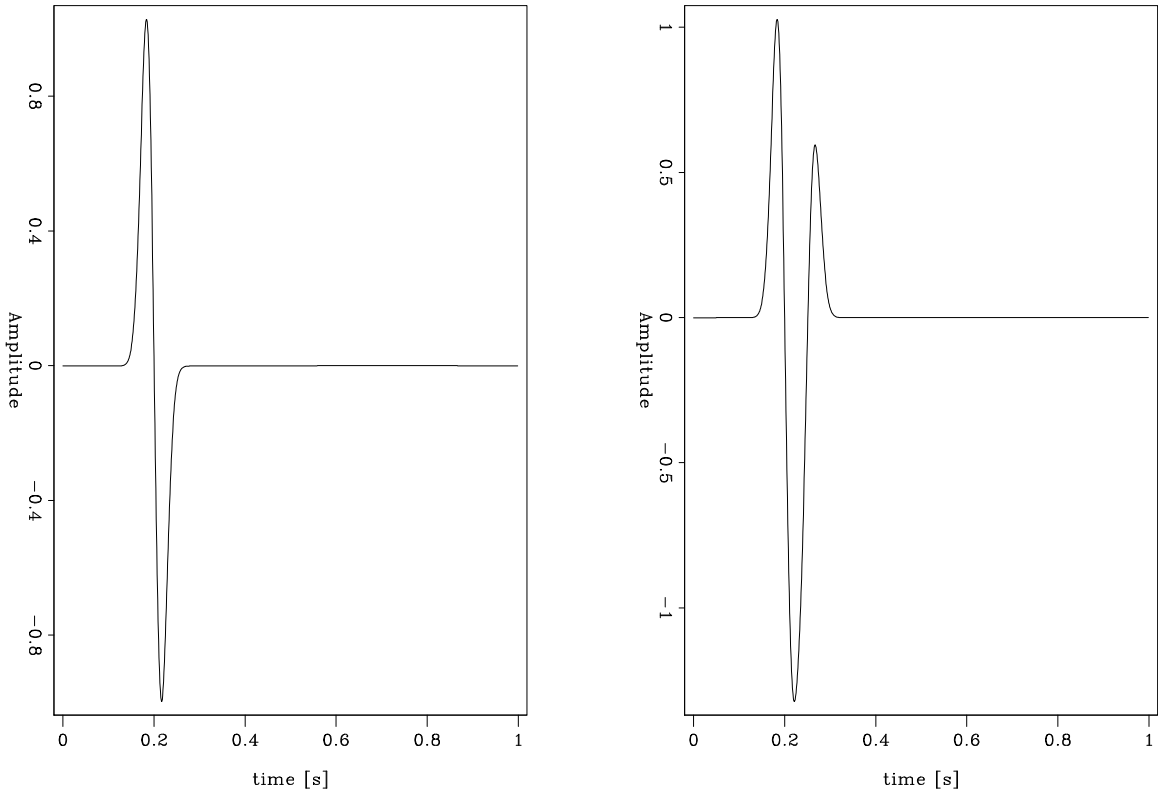


Figure 8: (left) Original unghosted trace for the second synthetic case. (right) Ghosted trace with ghosting parameters equal to $R = 0.6$ and $\tau = 50ms$. [**ER**]

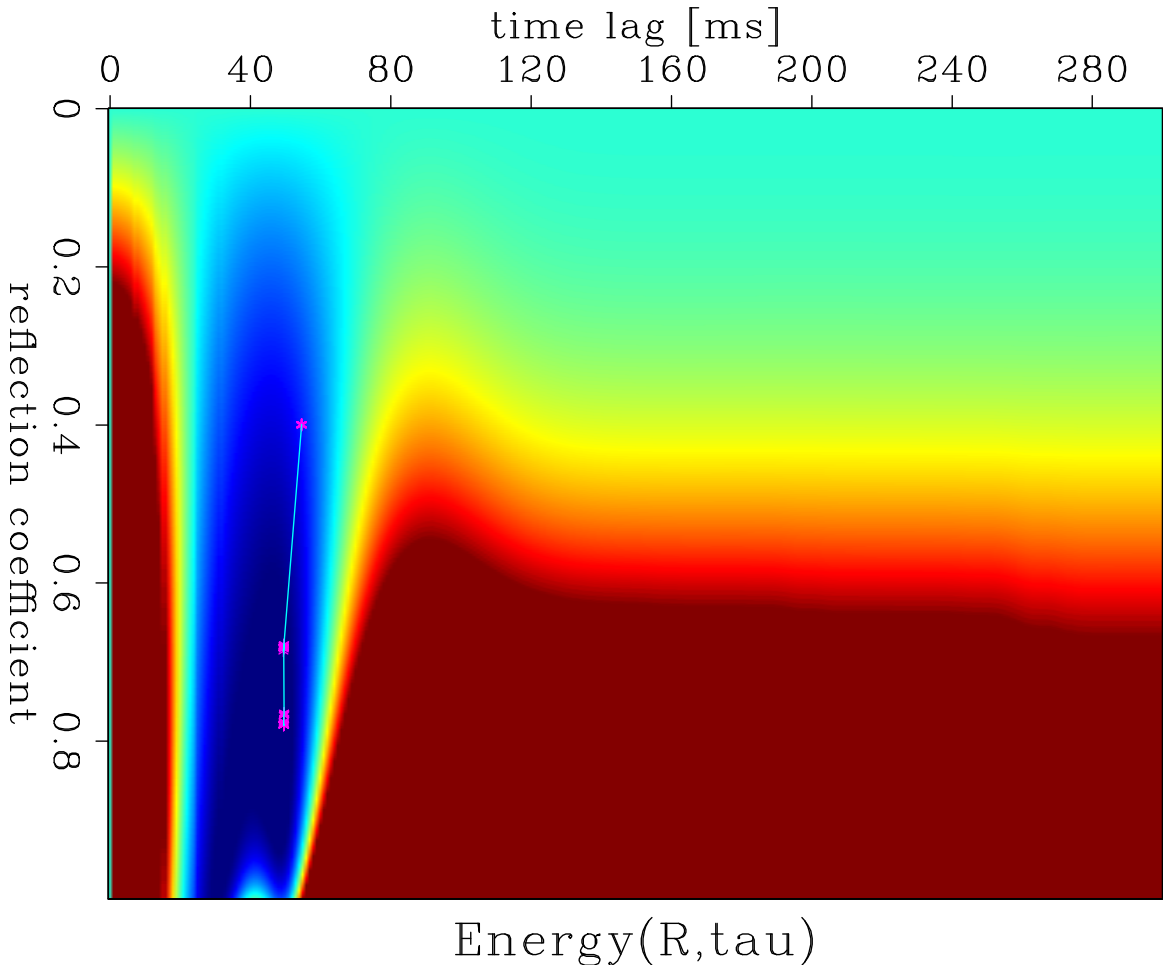


Figure 9: Same plot of Figure 6 but for the ghosted trace on the right panel of Figure 8. [**ER**]

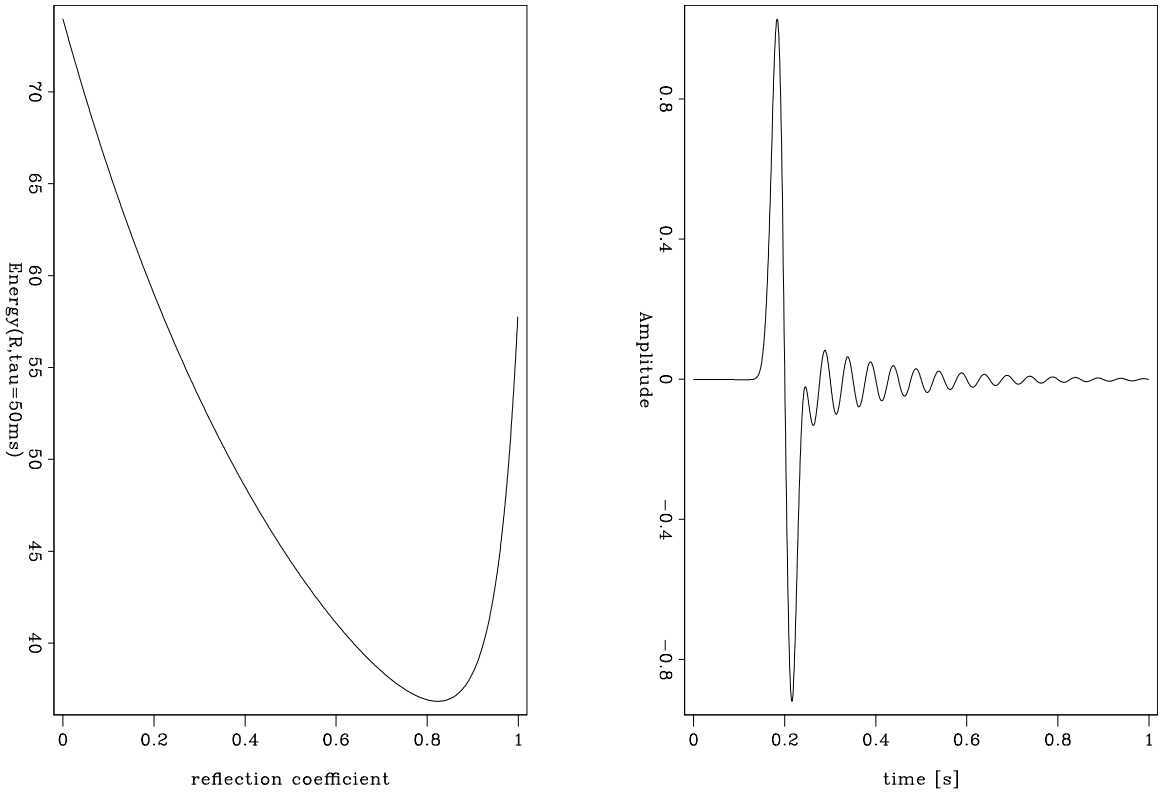


Figure 10: Section of the objective function of left panel of Figure 9 for $\tau = 50ms$. (right) Deghosted trace using the optimized parameters from the solver ($R = 0.78$ and $\tau = 49.95ms$). [**ER**]

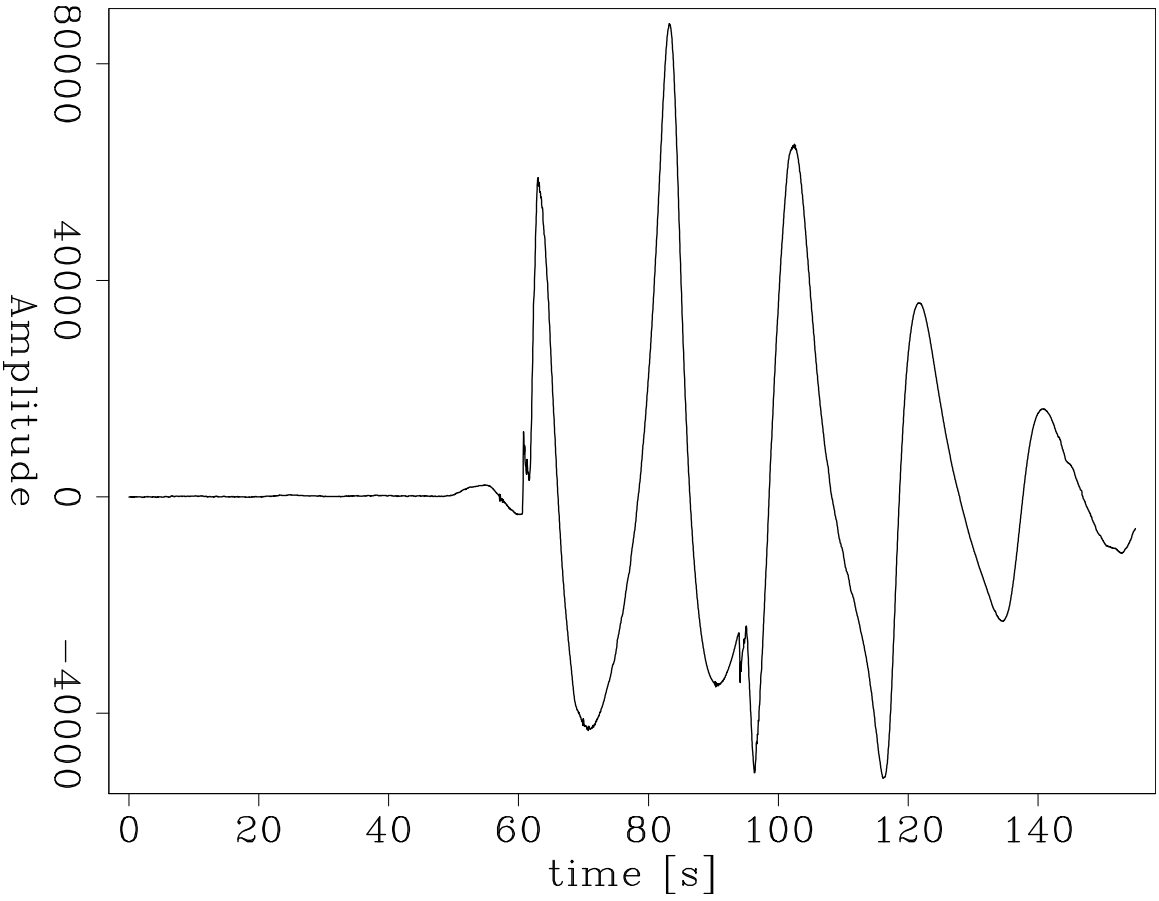


Figure 11: Real trace recorded during a airgun testing. The ghost delay is easily detectable and is approximately $33ms$. [**ER**]

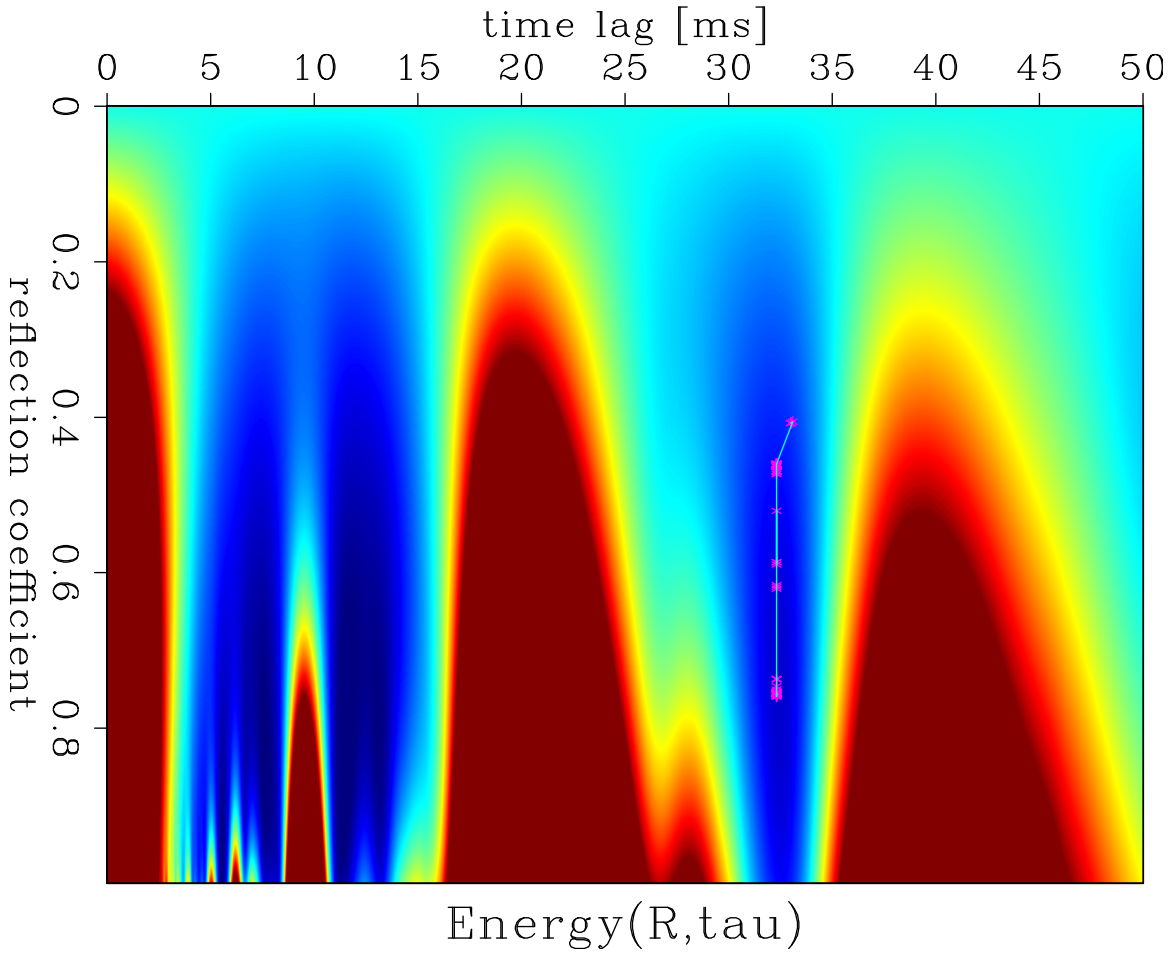


Figure 12: Energy plot and optimization path for the real trace test. [**ER**]

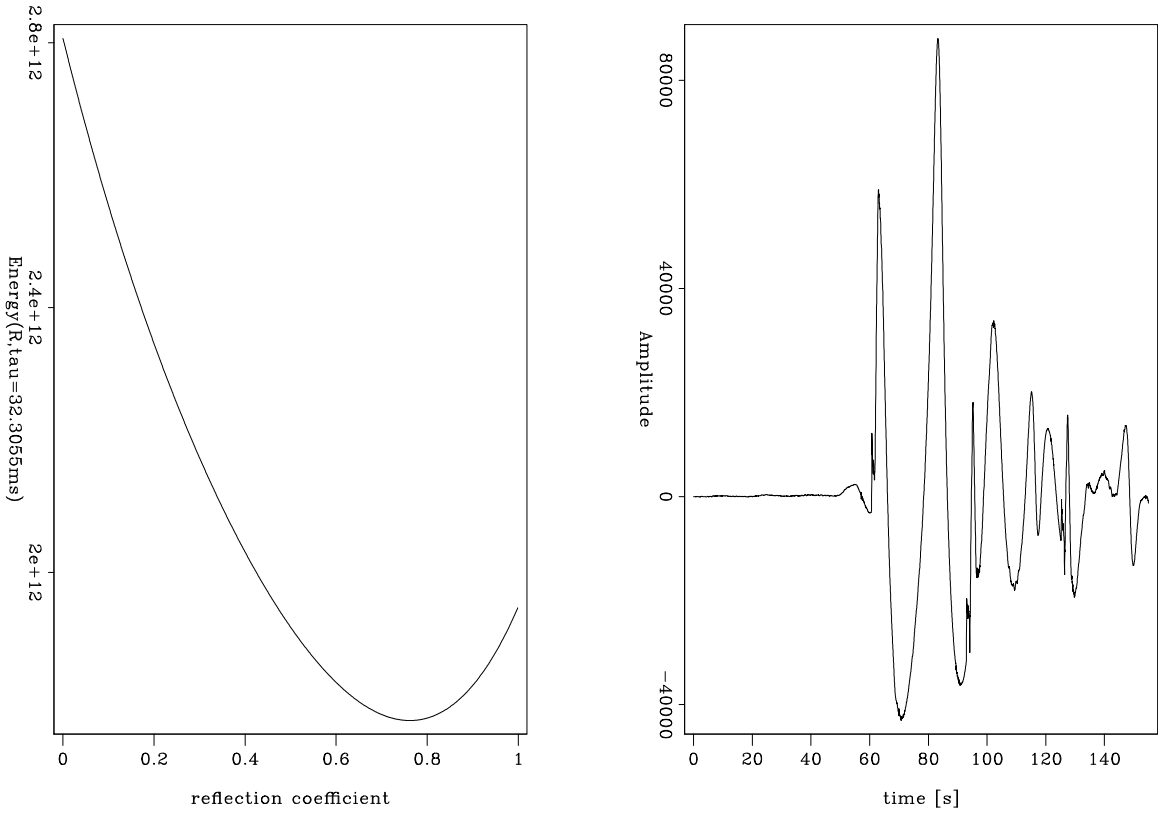


Figure 13: Section of the objective function of left panel of Figure 12 for $\tau = 32.3055ms$. (right) Deghosted trace using the optimized parameters from the solver ($R = 0.7564650$ and $\tau = 32.30554ms$). [**ER**]