PDE-constrained optimization with Alya

Oscar Peredo

CASE Department
Barcelona Supercomputing Center

May, 2013

Index

PDE-constrained optimization

Discrete adjoint method (reduced gradient)

Implementation using Alya

- Adding gradient calculation
- Adding optimization service

Example

Index

PDE-constrained optimization

Discrete adjoint method (reduced gradient)

Implementation using Alya

- Adding gradient calculation
- Adding optimization service

Example

PDE-constrained optimization

$$\begin{aligned} \underset{(u,d)\in\mathcal{U}\times\mathcal{D}}{\text{minimize}} \quad & \mathcal{J}(u,d) \\ \text{subject to} \quad & \mathcal{R}(u,d) = 0 \end{aligned} \qquad \text{(PDECO)}$$

with

- $\mathcal{U}$ and $\mathcal{D}$ assumed to be appropiate Hilbert functional spaces.
- $\mathcal{J} : \mathcal{U} \times \mathcal{D} \to \mathbb{R}$ the cost functional and $\mathcal{R} : \mathcal{U} \times \mathcal{D} \to \mathcal{U}$ the constraint operator, which is a PDE defined in a domain $\Omega \subset \mathbb{K}^n$ ($\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$ and $n \in \{1, 2, 3\}$).
- $u \in \mathcal{U}$ is the solution of the PDE.
- $d \in \mathcal{D}$ is the model or design parameter function which acts as an input of the PDE.

Examples

- Shape optimization in computational fluid dynamics.
- Material inversion in geophysics.
- Data assimilation in weather prediction modeling.
- Structural optimization of stressed systems.
- Control of chemical processes.
- Bio-engineering techniques in cancer treatment.
- Option pricing in computational finance.
- ...

Index

PDE-constrained optimization

Discrete adjoint method (reduced gradient)

Implementation using Alya
- Adding gradient calculation
- Adding optimization service

Example

Discrete adjoint method (reduced gradient)

Domain discretization + FEM:

$$\mathcal{R}(u,d) = 0 \qquad \Leftrightarrow \qquad \underbrace{\mathbf{A}(\mathbf{d})\mathbf{u} = \mathbf{b}(\mathbf{d})}_{\mathbf{R}(\mathbf{u},\mathbf{d})=\mathbf{0}}$$

Discrete version of (PDECO):

$$\begin{array}{lrcl} \underset{(\mathbf{u},\mathbf{d})\in\mathbb{K}^{n_u}\times\mathbb{R}^{n_d}}{\text{minimize}} & J(\mathbf{u},\mathbf{d}) & & \\ \text{subject to} & \mathbf{R}(\mathbf{u},\mathbf{d}) & = & \mathbf{0} \end{array} \qquad \text{(D-PDECO)}$$

Reduced version of (D-PDECO):

$$\underset{\mathbf{d}\in\mathbb{R}^{n_d}}{\text{minimize}} \quad j(\mathbf{d}) := J(\underbrace{\mathbf{u}(\mathbf{d})}_{\mathbf{A}(\mathbf{d})^{-1}\mathbf{b}(\mathbf{d})}, \mathbf{d}) \qquad \text{(R-D-PDECO)}$$

Discrete adjoint method (reduced gradient)

Calculation of $\nabla_{\mathbf{d}} j(\mathbf{d})$:

0. Set initial value of $\mathbf{d}$.
1. $\mathbf{u} \leftarrow \mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}$ (*forward problem*).
2. Calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and $\nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u}, \mathbf{d})$[1].
3. $\boldsymbol{\lambda} \leftarrow \nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u}, \mathbf{d})^T \boldsymbol{\lambda} = \nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})^T$ (*adjoint problem*).
4. Calculate $\nabla_{\mathbf{d}} J(\mathbf{u}, \mathbf{d})$ and $\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d})$[1].
5. $\nabla_{\mathbf{d}} j(\mathbf{d}) = -\boldsymbol{\lambda}^T \nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) + \nabla_{\mathbf{d}} J(\mathbf{u}, \mathbf{d})$.

[1] Using automatic differentiation, finite differences or taking derivatives *by hand*

Index

PDE-constrained optimization

Discrete adjoint method (reduced gradient)

Implementation using Alya

- Adding gradient calculation
- Adding optimization service

Example

Index

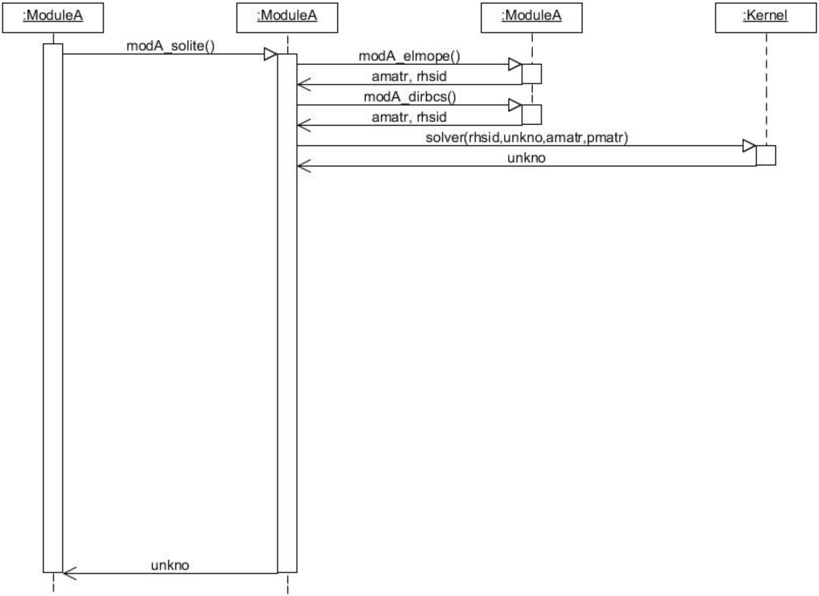
PDE-constrained optimization

Discrete adjoint method (reduced gradient)

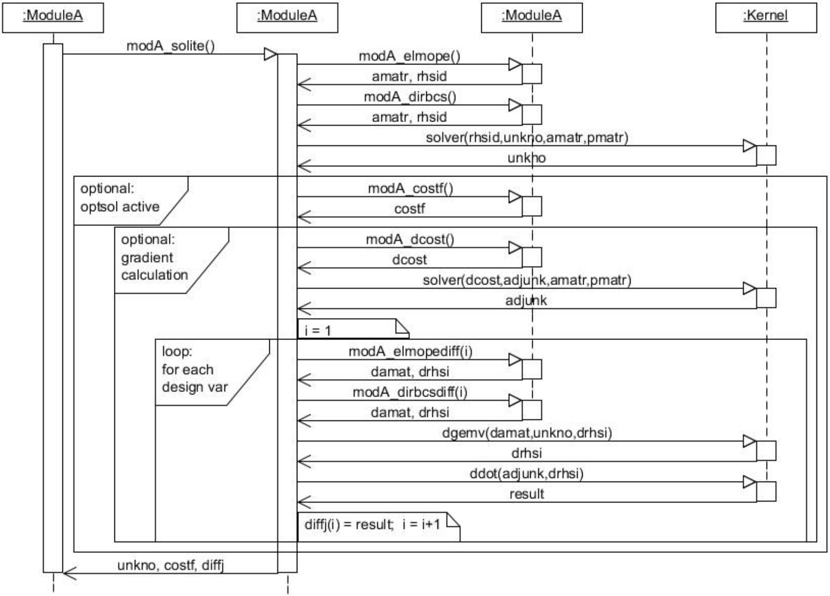
Implementation using Alya
- Adding gradient calculation
- Adding optimization service

Example

modA_solite.f90



modA_solite.f90 + gradient calculation



modA_solite.f90 + gradient calculation

- modA_costf: for each slave, calculate $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}), \mathbf{d})$, then apply MPI_AllReduce, and store it in scalar variable costf.
- modA_dcost: for each slave, calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and store it in (distributed) vector variable dcost (same size as rhsid).
- modA_elmopediff(i): for each slave, assemble $\frac{\partial \mathbf{A}(\mathbf{d})}{\partial \mathbf{d}_i}$ and $\frac{\partial \mathbf{b}(\mathbf{d})}{\partial \mathbf{d}_i}$, and store them in (distributed) vector variables damat and drhsi.
- modA_dirbcsdiff(i): for each slave, apply $\frac{\partial}{\partial \mathbf{d}_i}$ in b.c.'s and update damat and drhsi.
- dgemv(damat,unkno,drhsi): calculates

$$\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) = (\nabla_{\mathbf{d}} \mathbf{A}(\mathbf{d}))\,\mathbf{u} - \nabla_{\mathbf{d}} \mathbf{b}(\mathbf{d})$$

  using the expression $\mathtt{drhsi} \leftarrow \mathtt{damat} * \mathtt{unkno} - \mathtt{drhsi}$.
- ddot(adjunk,drhsi): calculates $\frac{\partial j(\mathbf{d})}{\partial \mathbf{d}_i}$ using the expression $\mathtt{result} \leftarrow \mathtt{adjunk}^T * \mathtt{drhsi}$.

`modA_solite.f90` + gradient calculation

- `modA_costf`: for each slave, calculate $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}), \mathbf{d})$, then apply `MPI_AllReduce`, and store it in scalar variable `costf`.
- `modA_dcost`: for each slave, calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and store it in (distributed) vector variable `dcost` (same size as `rhsid`).
- `modA_elmopediff(i)`: for each slave, assemble $\frac{\partial \mathbf{A}(\mathbf{d})}{\partial \mathbf{d}_i}$ and $\frac{\partial \mathbf{b}(\mathbf{d})}{\partial \mathbf{d}_i}$, and store them in (distributed) vector variables `damat` and `drhsi`.
- `modA_dirbcsdiff(i)`: for each slave, apply $\frac{\partial}{\partial \mathbf{d}_i}$ in b.c.'s and update `damat` and `drhsi`.
- `dgemv(damat,unkno,drhsi)`: calculates

  $$\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) = (\nabla_{\mathbf{d}} \mathbf{A}(\mathbf{d}))\, \mathbf{u} - \nabla_{\mathbf{d}} \mathbf{b}(\mathbf{d})$$

  using the expression $\mathtt{drhsi} \leftarrow \mathtt{damat} * \mathtt{unkno} - \mathtt{drhsi}$.
- `ddot(adjunk,drhsi)`: calculates $\frac{\partial j(\mathbf{d})}{\partial \mathbf{d}_i}$ using the expression $\mathtt{result} \leftarrow \mathtt{adjunk}^T * \mathtt{drhsi}$.

modA_solite.f90 + gradient calculation

- modA_costf: for each slave, calculate $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}), \mathbf{d})$, then apply MPI_AllReduce, and store it in scalar variable costf.
- modA_dcost: for each slave, calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and store it in (distributed) vector variable dcost (same size as rhsid).
- modA_elmopediff(i): for each slave, assemble $\frac{\partial \mathbf{A}(\mathbf{d})}{\partial \mathbf{d}_i}$ and $\frac{\partial \mathbf{b}(\mathbf{d})}{\partial \mathbf{d}_i}$, and store them in (distributed) vector variables damat and drhsi.
- modA_dirbcsdiff(i): for each slave, apply $\frac{\partial}{\partial \mathbf{d}_i}$ in b.c.'s and update damat and drhsi.
- dgemv(damat,unkno,drhsi): calculates

$$\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) = (\nabla_{\mathbf{d}} \mathbf{A}(\mathbf{d}))\, \mathbf{u} - \nabla_{\mathbf{d}} \mathbf{b}(\mathbf{d})$$

  using the expression $\texttt{drhsi} \leftarrow \texttt{damat} * \texttt{unkno} - \texttt{drhsi}$.
- ddot(adjunk,drhsi): calculates $\frac{\partial j(\mathbf{d})}{\partial \mathbf{d}_i}$ using the expression $\texttt{result} \leftarrow \texttt{adjunk}^T * \texttt{drhsi}$.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

`modA_solite.f90` + gradient calculation

- `modA_costf`: for each slave, calculate $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}), \mathbf{d})$, then apply `MPI_AllReduce`, and store it in scalar variable `costf`.
- `modA_dcost`: for each slave, calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and store it in (distributed) vector variable `dcost` (same size as `rhsid`).
- `modA_elmopediff(i)`: for each slave, assemble $\frac{\partial \mathbf{A}(\mathbf{d})}{\partial \mathbf{d}_i}$ and $\frac{\partial \mathbf{b}(\mathbf{d})}{\partial \mathbf{d}_i}$, and store them in (distributed) vector variables `damat` and `drhsi`.
- `modA_dirbcsdiff(i)`: for each slave, apply $\frac{\partial}{\partial \mathbf{d}_i}$ in b.c.'s and update `damat` and `drhsi`.
- `dgemv(damat,unkno,drhsi)`: calculates

$$\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) = (\nabla_{\mathbf{d}} \mathbf{A}(\mathbf{d}))\, \mathbf{u} - \nabla_{\mathbf{d}} \mathbf{b}(\mathbf{d})$$

  using the expression `drhsi` $\leftarrow$ `damat` $*$ `unkno` $-$ `drhsi`.
- `ddot(adjunk,drhsi)`: calculates $\frac{\partial j(\mathbf{d})}{\partial \mathbf{d}_i}$ using the expression `result` $\leftarrow$ `adjunk`$^T *$ `drhsi`.

modA_solite.f90 + gradient calculation

- modA_costf: for each slave, calculate $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}), \mathbf{d})$, then apply MPI_AllReduce, and store it in scalar variable costf.
- modA_dcost: for each slave, calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and store it in (distributed) vector variable dcost (same size as rhsid).
- modA_elmopediff(i): for each slave, assemble $\frac{\partial \mathbf{A}(\mathbf{d})}{\partial \mathbf{d}_i}$ and $\frac{\partial \mathbf{b}(\mathbf{d})}{\partial \mathbf{d}_i}$, and store them in (distributed) vector variables damat and drhsi.
- modA_dirbcsdiff(i): for each slave, apply $\frac{\partial}{\partial \mathbf{d}_i}$ in b.c.'s and update damat and drhsi.
- dgemv(damat,unkno,drhsi): calculates

$$\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) = (\nabla_{\mathbf{d}} \mathbf{A}(\mathbf{d}))\, \mathbf{u} - \nabla_{\mathbf{d}} \mathbf{b}(\mathbf{d})$$

  using the expression $\mathtt{drhsi} \leftarrow \mathtt{damat} * \mathtt{unkno} - \mathtt{drhsi}$.
- ddot(adjunk,drhsi): calculates $\frac{\partial j(\mathbf{d})}{\partial \mathbf{d}_i}$ using the expression $\mathtt{result} \leftarrow \mathtt{adjunk}^T * \mathtt{drhsi}$.

modA_solite.f90 + gradient calculation

- modA_costf: for each slave, calculate $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}), \mathbf{d})$, then apply MPI_AllReduce, and store it in scalar variable costf.
- modA_dcost: for each slave, calculate $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$ and store it in (distributed) vector variable dcost (same size as rhsid).
- modA_elmopediff(i) : for each slave, assemble $\frac{\partial \mathbf{A}(\mathbf{d})}{\partial \mathbf{d}_i}$ and $\frac{\partial \mathbf{b}(\mathbf{d})}{\partial \mathbf{d}_i}$, and store them in (distributed) vector variables damat and drhsi.
- modA_dirbcsdiff(i) : for each slave, apply $\frac{\partial}{\partial \mathbf{d}_i}$ in b.c.'s and update damat and drhsi.
- dgemv(damat,unkno,drhsi) : calculates

  $$\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) = (\nabla_{\mathbf{d}} \mathbf{A}(\mathbf{d}))\,\mathbf{u} - \nabla_{\mathbf{d}} \mathbf{b}(\mathbf{d})$$

  using the expression drhsi $\leftarrow$ damat $*$ unkno $-$ drhsi.
- ddot(adjunk,drhsi) : calculates $\frac{\partial j(\mathbf{d})}{\partial \mathbf{d}_i}$ using the expression result $\leftarrow$ adjunk^T $*$ drhsi.

Index

PDE-constrained optimization

Discrete adjoint method (reduced gradient)

Implementation using Alya

- Adding gradient calculation
- Adding optimization service

Example

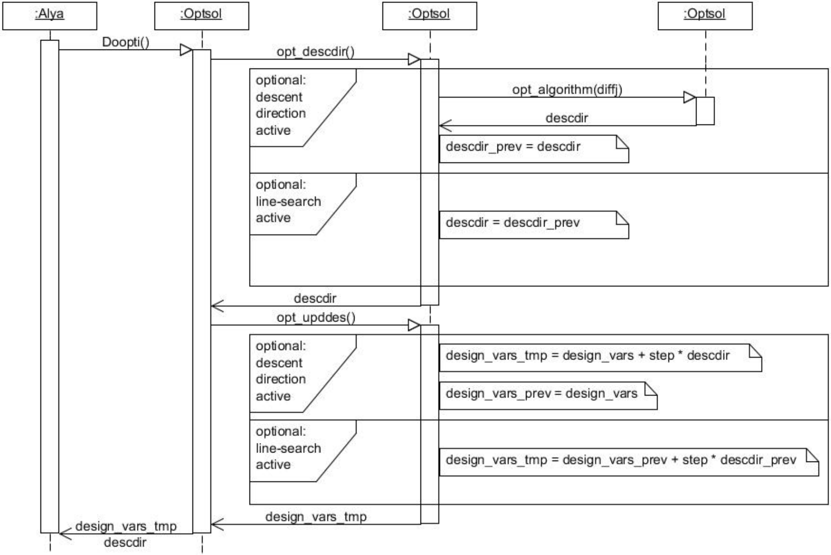
Alya.f90

```
program Alya

  call Turnon()

     call Iniunk()
     call Filter(ITASK_INITIA)
     call Output(ITASK_INITIA)
     time: do while(kfl_gotim==1)
        call Timste()
        call Begste()
        block: do while(kfl_goblk==1)
           coupling: do while(kfl_gocou==1)
              call Doiter()
              call Concou()
           end do coupling
           call Conblk()
           call Newmsh(one)
        end do block
        call Endste()
        call Filter(ITASK_ENDTIM)
        call Output(ITASK_ENDTIM)
     end do time
     call Filter(ITASK_ENDRUN)
     call Output(ITASK_ENDRUN)




  call Turnof()
end program Alya
```

Alya.f90 + optimization service

```
program Alya
   call Begopt()         <------------------------------ Initialization of opti params
   call Turnon()
   optimization: do while ( kfl_goopt == 1)
      call Iniunk()
      call Filter(ITASK_INITIA)
      call Output(ITASK_INITIA)
      time: do while(kfl_gotim==1)
         call Timste()
         call Begste()
         block: do while(kfl_goblk==1)
            coupling: do while(kfl_gocou==1)
               call Doiter()  <------------ Assembling, fwd/adj solver and diffj calculation
               call Concou()
            end do coupling
            call Conblk()
            call Newmsh(one)
         end do block
         call Endste()
         call Filter(ITASK_ENDTIM)
         call Output(ITASK_ENDTIM)
      end do time
      call Filter(ITASK_ENDRUN)
      call Output(ITASK_ENDRUN)
      call Doopti()   <-------------------------- Update design variables using grad from Doiter
      call Endopt()   <------------------------------ Check tolerance and max opti iters
   end do optimization
   call Output(ITASK_ENDOPT)
   call Turnof()
end program Alya
```

Doopti.f90


:Alya
:Optsol
:Optsol
:Optsol
Doopti()
opt_descdir()
optional: descent direction active
opt_algorithm(diffj)
descdir
descdir_prev = descdir
optional: line-search active
descdir = descdir_prev
descdir
opt_upddes()
optional: descent direction active
design_vars_tmp = design_vars + step * descdir
design_vars_prev = design_vars
optional: line-search active
design_vars_tmp = design_vars_prev + step * descdir_prev
design_vars_tmp
design_vars_tmp
descdir

Doopti.f90

- `opt_descdir`: calculate descent direction and store it in a vector variable `descdir`. If linesearch is active (`kfl_curlin_opt>1`), use desc. direction obtained in the first linesearch iteration.
- `opt_algorithm`: using `diffj` ($\nabla_{\mathbf{d}} j$) obtain descent direction `descdir` using some algorithm (steepest descent, conjugate gradient, quasi-Newton). Currently only steepest descent is programed.
- `opt_upddes`: update design variables `design_vars` and store the new value in a temporal variable `design_vars_tmp`.

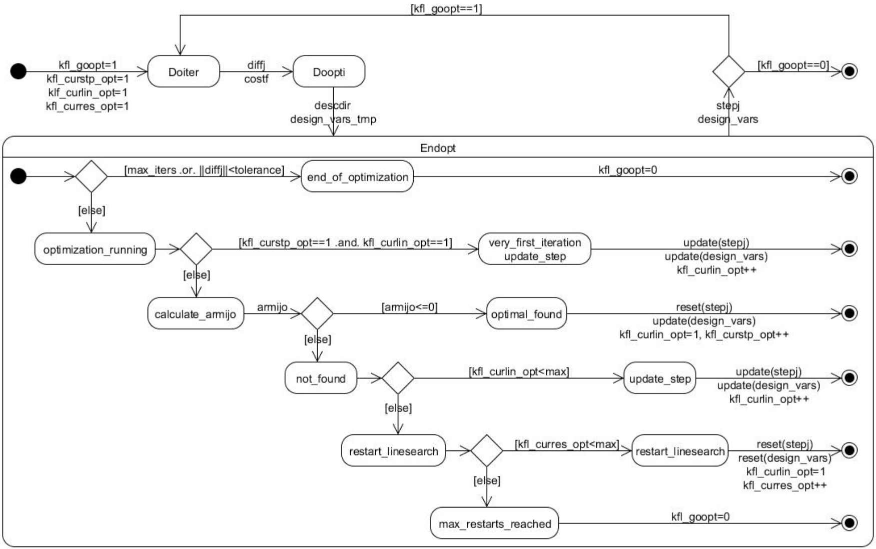
Doopti.f90

- opt_descdir: calculate descent direction and store it in a vector variable descdir. If linesearch is active (kfl_curlin_opt>1), use desc. direction obtained in the first linesearch iteration.
- opt_algorithm: using diffj ($\nabla_{\mathbf{d}} j$) obtain descent direction descdir using some algorithm (steepest descent, conjugate gradient, quasi-Newton). Currently only steepest descent is programed.
- opt_updes: update design variables design_vars and store the new value in a temporal variable design_vars_tmp.

Doopti.f90

- opt_descdir: calculate descent direction and store it in a vector variable descdir. If linesearch is active (kfl_curlin_opt>1), use desc. direction obtained in the first linesearch iteration.
- opt_algorithm: using diffj ($\nabla_{\mathbf{d}} j$) obtain descent direction descdir using some algorithm (steepest descent, conjugate gradient, quasi-Newton). Currently only steepest descent is programed.
- opt_upddes: update design variables design_vars and store the new value in a temporal variable design_vars_tmp.

Endopt.f90



[kfl_goopt==1]
kfl_goopt=1
kfl_curstp_opt=1
klf_curlin_opt=1
kfl_curres_opt=1
Doiter
diffj
costf
Doopti
[kfl_goopt==0]
descdir
design_vars_tmp
stepj
design_vars
Endopt
[max_iters .or. ||diffj||<tolerance]
end_of_optimization
kfl_goopt=0
[else]
optimization_running
[kfl_curstp_opt==1 .and. kfl_curlin_opt==1]
very_first_iteration
update_step
update(stepj)
update(design_vars)
kfl_curlin_opt++
[else]
calculate_armijo
armijo
[armijo<=0]
optimal_found
reset(stepj)
update(design_vars)
kfl_curlin_opt=1, kfl_curstp_opt++
[else]
not_found
[kfl_curlin_opt<max]
update_step
update(stepj)
update(design_vars)
kfl_curlin_opt++
[else]
restart_linesearch
[kfl_curres_opt<max]
restart_linesearch
reset(stepj)
reset(design_vars)
kfl_curlin_opt=1
kfl_curres_opt++
[else]
max_restarts_reached
kfl_goopt=0

Index

PDE-constrained optimization

Discrete adjoint method (reduced gradient)

Implementation using Alya

- Adding gradient calculation
- Adding optimization service

Example

Inverse 3D Controlled-source Electromagnetism

Secondary potential formulation of Maxwell's equations,

$$\nabla^2 \mathbf{A}_s + i\omega\mu_0\sigma(\mathbf{A}_s + \nabla\phi_s) = -i\omega\mu_0\delta\sigma(\mathbf{A}_p + \nabla\phi_p) \tag{1}$$

$$\nabla \cdot (i\omega\mu_0\sigma(\mathbf{A}_s + \nabla\phi_s)) = -\nabla \cdot (i\omega\mu_0\delta\sigma(\mathbf{A}_p + \nabla\phi_p)) \tag{2}$$

discretized PDE + boundary conditions:

$$\boldsymbol{K}(\boldsymbol{d})\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{d}) \tag{3}$$

with $\boldsymbol{z} = (\mathbf{A}_s^x, \mathbf{A}_s^y, \mathbf{A}_s^z, \phi_s) \in \mathbb{C}^{4n}$
In Alya, module `Helmoz` (developed by Jelena and Vladimir).
Cost function:

$$J(\boldsymbol{z}, \boldsymbol{d}) \;=\; \overline{(\boldsymbol{z} - \boldsymbol{z}^{obs})}^T \boldsymbol{M}^{obs}(\boldsymbol{z} - \boldsymbol{z}^{obs}) \tag{4}$$

Inverse 3D Controlled-source Electromagnetism

- Currently, we have implemented:
  - `hlm_costf`: 2 types
  - `hlm_dcost`: 2 types
  - `hlm_elmopediff`, `hlm_dirbcsdiff`: our design variable is the isotropic electric conductivity tensor (# design vars = number of nodes in the mesh)
  - `hlm_solite`: new features added (multiple shots, optimization in some routines to speed up the exec time, ...)
- We are still in the gradient obtention stage (very difficult).
- After this, when we have a *good* gradient at our disposal, we can calculate a descent direction, linesearch, ...

hlm_solite.f90

```
subroutine hlm_solite()

     call hlm_matrix()
     call solvex(rhsix,unknx,amatx,pmatx)

























end subroutine hlm_solite
```

hlm_solite.f90 + gradient calculation

```fortran
subroutine hlm_solite()
  shots: do ishot=1,nshot_hlm
     call hlm_matrix()
     call solvex(rhsix,unknx,amatx,pmatx)
     if(kfl_servi(ID_OPTSOL)==1) then
        if( INOTMASTER ) then
           call hlm_costev(delta,shot)
           if(kfl_curlin_opt==1)then
              call hlm_dcost(delta,shot)
           end if
        end if
        call pararr('SUM',0_ip,1_ip,costf_shot)
        if(kfl_curlin_opt==1) then
           call hlm_adjvar()
           do indvars=1,kfl_ndvars_opt
              if(INOTMASTER) then
                 call hlm_elmopediff(indvars)
                 call hlm_dirbcsdiff(indvars)
              end if
              call bcsplx2( npoin, 4_ip, damat, c_sol, r_sol, unknx, drhsi)
              call proptx(npoin,4_ip,aunknx,drhsi,rr)
              diffj_shot(indvars) = 2.0_rp * real(rr)
           end do
        end if
     costf=costf + costf_shot
     diffj = diffj + diffj_shot
     end if
  end do shots
end subroutine hlm_solite
```

Thanks for your attention!