Inverse problems

High-order optimization and parallel computing Lecture 7

Nikolai Piskunov
2014

Non-linear least square fit

The (conjugate) gradient search has one important problem which often occurs during Least Square Fit (LSF): the value of discrepancy Ω has not enough dynamic range to hold the contribution from all data points. This can be illustrated using the DI problem: the elements below the equator have significant contribution in just a few phases but when all wavelength and phases are combined, the contribution of such elements may be vanishingly small.

- One could notice that the corresponding components of the gradient are small but not negligible while the Ω does not change if we modify (a little) the corresponding surface elements.
- In case when we compare the observations with synthetic values the discrepancy does not have to be characterized by a single number. Instead we can try to keep the differences in each individual data point (λ, ϕ, in case of DI) in a single vector.

Second order optimization

$$\Omega = \sum \left[g_i^{\text{Calc}}(\vec{f}) - g_i^{\text{Obs}} \right]^2 = \min$$

$$\frac{\partial \Omega}{\partial f_j} = 2 \sum_i \left[g_i^{\text{Calc}}(\vec{f}) - g_i^{\text{Obs}} \right] \frac{\partial g_i^{\text{Calc}}}{\partial f_j}$$

$$\frac{\partial^2 \Omega}{\partial f_j \partial f_k} = 2 \sum_i \left\{ \frac{\partial g_i^{\text{Calc}}}{\partial f_j} \frac{\partial g_i^{\text{Calc}}}{\partial f_k} + \left[g_i^{\text{Calc}} - g_i^{\text{Obs}} \right] \frac{\partial^2 g_i^{\text{Calc}}}{\partial f_j \partial f_k} \right\}$$

$$\left. \frac{\partial \Omega}{\partial f_j} \right|_{\min} = 0 \Rightarrow \frac{1}{2} \sum_k \frac{\partial^2 \Omega}{\partial f_j \partial f_k} \Delta f_k = -\frac{1}{2} \frac{\partial \Omega}{\partial f_j}$$

- If we can find a way to compute second derivatives we should be able to obtain the corrections to the *f* 's and we don't need to search for the step size!
- In practice this is close to impossible:
  - Very seldom we can compute second derivative analytically, approximate calculations are major source of numerical errors;
  - For large number of free parameters the size of the matrices becomes prohibitively large.

Levenberg-Marquardt method

Compare the quadratic approximation:

$$\sum_k \mathbf{A}_{jk} \Delta f_k = \mathbf{b}_j$$

with the gradient search:

$$\frac{1}{step} \times \Delta f_j = \mathbf{b}_j$$

where:

$$\mathbf{A}_{jk} = \frac{1}{2} \frac{\partial^2 \Omega}{\partial f_j \partial f_k}, \qquad \mathbf{b}_j = -\frac{1}{2} \frac{\partial \Omega}{\partial f_j}$$

The expression for the second derivatives consists of two parts:

$$\mathbf{A}_{jk} = \sum_i \left\{ \frac{\partial g_i^{\text{Calc}}}{\partial f_j} \frac{\partial g_i^{\text{Calc}}}{\partial f_k} + \left[g_i^{\text{Calc}} - g_i^{\text{Obs}} \right] \frac{\partial^2 g_i^{\text{Calc}}}{\partial f_j \partial f_k} \right\}$$

The first part is nothing new compared to the gradient search. The result is a symmetric matrix. The second part is the main source of numerical errors but it will become negligible close to the minimum.

- We will ignore the second derivatives of g's as the main source of numerical errors assuming that out model calculations is realistic enough and close to minimum the second part (summed over all observations) will vanish anyway.
- Far from minimum our quadratic approximation is probably bad and the simple gradient search seems to converge robustly there, so let's try to combine the two methods with a smooth transition from gradient search far from the minimum to the quadratic method close in.

First, let us write the two methods in the matrix form:

$$\sum_k \mathbf{A}_{jk} \Delta f_k = \mathbf{b}_j$$

$$c \cdot \sum_k \delta_{jk} \Delta f_k = \mathbf{b}_j$$

$$\frac{1}{2} \sum_k \left(\mathbf{A}_{jk} + c \cdot \delta_{jk} \right) \Delta f_k = \mathbf{b}_j$$

In fact, this will work better if would use additional information about the step size along different directions:

$$\begin{pmatrix} a_{11}(1+c) & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22}(1+c) & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN}(1+c) \end{pmatrix} \cdot \begin{pmatrix} \Delta f_1 \\ \Delta f_2 \\ \vdots \\ \Delta f_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

- With large c the system turns to gradient search while for small $\boldsymbol{c}$ it follows the 2nd order approximation.
- Clever way of adjusting $\boldsymbol{c}$ is as following:

1° Select the initial (large) value of $\boldsymbol{c}$
2° Compute gradient of Ω
3° Construct matrix $\mathbf{A}_{jk}$ and RHS $\mathbf{b}_{jk}$
4° Try smaller/larger $\boldsymbol{c}$ - whichever decreases Ω
5° Once optimal $\boldsymbol{c}$ is found, re-compute gradient and continue from 3°

- In practice, there is more than one way of selecting parameter ***c*** and, therefore, calculating the next step. Often it is worth adjusting ***c*** as we go: compute gradient, construct $\mathbf{A}_{jk}$ and keep adjusting ***c*** until minimum of Ω is found.
- The matrix $\mathbf{A}_{jk}$ is symmetric. LU decomposition can be stored in place of the original matrix and allows for quick solution. Thus for the adjustment of ***c*** we don't need to solve the whole system of linear equations from scratch or even to restore the original matrix.

Parallelization

- MPI – Message Passing Interface
- MPI is better suited for problems where individual processors need access to a small fraction of the computational domain and communication takes small time compared to the calculations
- MPI allows each processor to figure out its number and perform two type of communications: *processor to processor* and *processor to all processors*
- Two models: *Single-Program-Multiple-Data* and *Multiple-Program-Multiple-Data*

Gradient of Ω

Partial derivative of Ω over abundance in surface element u,v (latitude, longitude) is:

$$\frac{\partial \Omega}{\partial Z_{uv}} = 2 \cdot \sum_{\lambda\phi} \left[R_{\lambda\phi}^{\text{Calc}} - R_{\lambda\phi}^{\text{Obs}} \right] \frac{\partial R_{\lambda\phi}^{\text{Calc}}}{\partial Z_{uv}} +$$

$$+2\Lambda \cdot (4Z_{uv} - Z_{u-1v} - Z_{u+1v} - Z_{uv-1} - Z_{uv+1})$$

and the derivative of the flux is given by:

$$\frac{\partial R_{\lambda\phi}^{\text{Calc}}}{\partial Z_{uv}} = \mu \cdot \frac{dI_{\lambda+\Delta\lambda}(u,v,\mu)}{dZ_{uv}} \Delta\sigma_{uv}$$

Surface element

Numerical aspects

- The convolution with the instrumental profile can be interchanged with the disk integration
- The gradient can be computed in the same loop as the flux:

loop over u :
  loop over v:

$$F_{\lambda\phi} = F_{\lambda\phi} + \mu I_{\lambda+\Delta\lambda}(u, v, \mu)\Delta\sigma$$

$$\frac{\partial R_{\lambda\phi}^{\text{Calc}}}{\partial Z_{uv}} = \mu \cdot \frac{dI_{\lambda+\Delta\lambda}(u, v, \mu)}{dZ_{uv}} \Delta\sigma_{uv}$$

FORTRAN implementation SPMD

```
      NTOT=<total job size>
C
C Initialization/communication section
C
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, N_PROC,IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, MY_PROC,IERR)
      NCHUNK=NTOTAL/N_PROC
C
C Receive from previous processor modify and send to the next
C
      IF(MY_PROC.GT.0) THEN
        CALL MPI_Recv(MY_PROC-1,1,MPI_DOUBLE,VAR,101,MPI_COMM_WORLD,IE)
        VAR=func(VAR,NCHUNK)
      ELSE
        VAR=0.D0
      ENDIF
      IF(MY_PROC.LT.N_PROC-1) THEN
        CALL MPI_Send(MY_PROC+1,1,MPI_DOUBLE,VAR,101,MPI_COMM_WORLD,IE)
      ENDDO
  …
      CALL MPI_FINALIZE(IERR)
```

Message passing

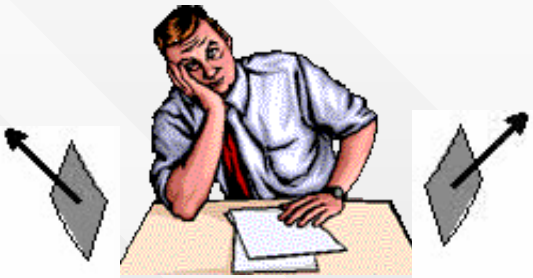
- Communication sends one variable at the time (scalar or array)
- If the receiving process is not ready, sending process waits (blocking communication)
- Data is temporary stored in a buffer – one for each computer
- Works well on a single machine with multiple processors or many machines

MPMD

- Much more complicated
- Simplest incarnation is Master-Worker(s)
- Make sense when you can distribute “job order” out-of-order
- One can use “non-blocking” communication and the synchronization mechanisms

Parallelization

(automatic load balance)



Optional:

- For those who are interested I have a demo implementation for MPMD in form of two FORTRAN programs m.f and p.f
- You can look at them - they are very minimalistic - compile and run them on any Linux machine or Mac with MPI installed.