Code-flow for joint inversion of radar and electrical resistivity

Diego Domenzain

1 Overview

This document explains the code-flow for a joint inversion of *ground penetrating radar* (GPR) and *electrical resistivity* (ER) data using the package **gerjoii**.

The name **gerjoii** stands for **g**round penetrating radar and **e**lectrical **r**esistivity **jo**int **i**nversion and **i**nterferometry.

Our inversion recovers electrical permittivity (ε) and conductivity (σ) of the subsurface. It assumes a 2.5D subsurface geometry and uses gradient-descent.

Job submission is done by shell scripts, and visualization is done with Python. The heavy-lifting code is written in Matlab.

2 Inside *gerjoii*

The main directory **gerjoii** has sub-directories `data, docs, field` and `src`.

Projects for specific data-sets live in `field`. Inside a particular project in `field` there are sub-directories where inversions with different hyper-parameters were done.

For example, the directory `field/groningen2-fi/g0/` holds the needed scripts for running an inversion on the Groningen data.

Inside each `g#` there is a directory `scripts`. This directory has Matlab scripts that set-up global variables needed in the inversion.

The next section details what happens next. We go through an inversion-flow whose data claims a very large domain, so we divide it in tiny sub-domains.

3 Down the rabbit hole - Matlab

Assume we are in `field/groningen2-fi/g0/scripts/`. The inversion has the following code-flow:

1. **Run the inversion with** `wdc_begin_disk_.m`,
    - load the inversion hyper-parameters,
    - load the initial ε and σ models,
    - build the tiny sub-domains,
    - load the parallel profile for Matlab,
    - perform the inversion (`wdc_imageOBJe2_5d_disk.m`)
    - save the results of the inversion.
2. **The inversion** `wdc_imageOBJe2_5d_disk.m`. A while-loop that iterates the following,
    - Compute cross-gradient updates for σ. Not computationally expensive and not done in parallel.
    - Compute updates for σ using the GPR data. Depending on the choice of hyper-parameters, this is done in one of two ways: `w_update_s_disk_.m` or `w_updateOBJ_s_disk_.m`. We only review the former,
        - `w_update_s_disk_.m` begins a Matlab `parfor` that runs on the number of experiments of the data. The `parfor` runs `w_update_s_disk.m`,
            * Loads data on a tiny version of the entire 2D domain. Every step that follows uses just this tiny sub-domain.
            * Solves the GPR forward model, that is Maxwell's 2D wave equation using a finite difference time domain solver. It also saves the entire wavefield (up to 20Gb), a matrix of dimensions length $\times$ depth $\times$ time. This step is CPU and memory intensive.
            * Solves the GPR forward model using the residual as source, and at each time-step cross-correlates with the recorded wavefield. Once this step is done, the wavefield is deleted. The result of the cross-correlation is the sought update for our gradient-descent.
            * Finds a step-size for this gradient. The forward model needs to be solved again. It is now an update.
            * Saves this update (which is the size of the tiny domain) in disk.
        - Read all updates that were saved to disk and sum them up to get an update on the entire domain. We now have a GPR update for σ.
    - Compute updates for σ using the ER data, `dc_update2_5d__.m`.

- `dc_update2_5d__.m` begins a Matlab `parfor` that runs on the number of experiments of the data. The `parfor` runs `dc_update2_5d.m`,
  - Loads data in the entire 2D domain.
  - Solves the ER forward model, that is Maxwell's 2.5D steady state equation with `dc_fwd2_5d.m`,
    - `dc_fwd2_5d.m` begins a for-loop of 4 iterations. Each iteration solves and saves in memory a 2D electric potential. These forward models consist of a linear solver and are not CPU nor memory intensive.
    - Once finished it performs a linear combination of these 2D electric potentials to give one 2.5D electric potential. The 2D electric potentials need to remain saved in memory.
  - `dc_gradient2_5d.m` computes the 2.5D gradient,
    - `dc_gradient2_5d.m` begins a for-loop of 4 iterations. Each iteration solves the ER forward model using the residuals as 2D sources and then multiplying by a special sparse matrix that uses the 2D electric potentials stored in memory.
    - Once finished it performs a linear combination of these 2D gradients to give one 2.5D gradient. The 2D electric potentials are deleted.
  - Find a step size for the gradient. This entails solving the 2.5D forward model again for different values of σ, so all the parts from `dc_fwd2_5d.m` are repeated. The gradient is now an update.
  - Save the update in memory.
- Sum all updates from all source-receiver pairs together. We now have an ER update for σ.

- Join σ updates from GPR and ER in a special way. This step is not computationally expensive and not done in parallel.
- Compute cross-gradient updates for ε. Not computationally expensive and not done in parallel.
- Compute updates for ε using the GPR data. Depending on the choice of hyper-parameters, this is done in one of two ways: `w_update_e_disk_.m` or `w_updateOBJ_e_disk_.m`. These are very similar to those for σ. The main difference is the number of times the forward model is called when finding the step-size.
- The while-loop is terminated when the number of iterations wanted is achieved.

4 Notes on time

Below is a table with approximate timings for the Groningen field data forward models. Each timing is per core per experiment, i.e. one shot-gather (GPR) on one core, or one

source-receiver pair (ER) on one core.

The times per iteration the forward models are called in the main while-loop (`wdc_imageOBJe2_5d_disk.m`) are also given. We assume we are using `w_update_[s,e]_disk_.m`. For `w_updateOBJ_[s,e]_disk_.m` the times the forward models are called is double.

forward model	time it takes	σ update	ε update	times used
2D GPR	20min	✓		2
2.5D ER	3min	✓	-	3
2D GPR	20min		✓	3

The total number of experiments in the GPR and ER data are 56 and 234 respectively.

Recall that `w_update_s_disk.m` demands 20Gb per experiment when computing the gradient. This means that out of the 56 experiments we can only do a fraction f at a time, so 20Gb $\times f$ does not exceed memory limits.