SnapVX Developers Guide

Abhijit Sharang
Stanford University
Stanford, CA 94305
abhisg@stanford.edu

David Hallac
Stanford University
Stanford, CA 94305
hallac@stanford.edu

This document is meant for developers interested in contributing to the code base. It provides instructions on how to submit bug reports, add new unit tests, contribute code, improve documentation, and more.

If you feel we are missing anything here, or if a section is unclear, please reach out to David Hallac at `hallac@stanford.edu` and we will be happy to help!

1 Introduction to SnapVX

SnapVX is a convex optimization solver for problems which are defined on a graph. Given a vertex set$\mathcal{V}$ and an edge set $\mathcal{E}$, we solve the following optimization problem:

$$\min \sum_{i \in \mathcal{V}} f_i(x_i) + \sum_{(j,k) \in \mathcal{E}} g_{jk}(x_j, x_k)$$

The variables are $x_1, x_2, .., x_m$. Here, $x_i \in \mathbb{R}^p$ is the variable and f_i is the cost function at node i.g_{jk} is the cost function associated with edge (j, k). When the cost functions are complex, SnapVX provides:

- A solution based on the alternating direction method of multipliers (ADMM) that is guaranteed to converge to the global optimum for any problem of this form.
- A fast, scalable algorithm which is capable of being distributed across multiple cores of a single machine.
- An integration of `Snap.py` and `CVXPY`.

2 Contributing to SnapVX

2.1 Submitting a bug report

In case you have any issues in using the package, please submit a ticket at the `https://github.com/snap-stanford/snapvx/issues`. You are also welcome to add new features and pull requests.

2.2 Fixing existing bugs

We keep track of all bugs at `https://github.com/snap-stanford/snapvx/issues`. If there is an open issue that no one has resolved, feel free to claim it on `Github` (by "assigning" it to yourself). Once it is fixed, see Section 2.4 for how to merge your modified code into the code base.

2.3 Retreiving the latest code

We use `git` for version control and host the repository on `Github`.

You can check out the latest sources with the command:
`git clone https://github.com/snap-stanford/snapvx.git`.

To generate a development version, run `setup.py` with `develop` flag:

```
python setup.py develop
```

If you do not have `sudo` access, you would have to add the `--user` flag to the above command as well. For more installation options, refer to the python `distutils` module.

Run the test script `Tests/test_script.sh` to check if the installation proceeded correctly.

2.4 Contributing new code

The preferred way to contribute to `snapvx` it to fork the main repository on Github and issue a pull request.

1. Create an account on `github` in case you don't already have one.
2. Fork the project repository by clicking on Fork button on the upper right corner of the repository's page.
3. Create a branch to incorporate your changes.
   `git checkout -b branch_name`
4. Write your tests inside the appropriate hierarchy of the `Tests` folder as explained in 2.6. (OPTIONAL, Highly RECOMMENDED)
5. Make sure the existing tests are passing. In case any test fails, use the error messages to fix the affected section of the code.
6. Commit your changes to `git`.
   `git add your_new_file(s) git commit`
7. Push these changes upstream.
   `git push -u origin branch_name`

Once you are satisfied with the changes you have made, issue a pull request by going to your repository's page `https://github.com/your_id/snapvx` and clicking on the button "New pull request"

2.5 Contributing examples

If you have new examples, feel free to add them to the Examples folder! Before you do so, please add comments to your code, so users understand how it works.

2.6 Testing and improving unit tests

We have divided the unit tests into three components, each catering to a different aspect of the package.

- `tests_installation` - This folder contains basic tests to check if the package and its dependencies have been installed correctly and the package is behaving expectedly.
- `tests_functionality` - All tests pertaining to different solver features that the package provides go here.
- `tests_scalability` - This contains tests for how well the package handles problems with different unknowns and graph sizes.

Running test suites requires python `unittest` module. To run all tests, execute the script `test_script.sh` in the `Tests` folder. This will display the details of all the tests, whether they passed or failed, and in the latter case, the specific reason behind the test failing.

You can add your own test inside the appropriate folder. Make sure that your python file is defined as `test_`<your test name>. Your test must define a class deriving `BaseTest` which is

present at the top hierarchy of the `Tests` folder (see the existing tests for sample syntax). Write the relevant unittest functions, and execute the shell script `test_script.sh` to check that the tests pass.

In case your test does not fit the test hierarchy we have defined, please create another folder with the name `tests_`<your feature name> and write your tests inside this folder in the same format as described above. Also make sure to edit the test script to look for tests inside this folder. Otherwise, if your tests fall into one of the three predefined folders, the test script will handle the new tests automatically.

3 Contributing to Documentation

3.1 User Documentation

You are welcome to contibute any form of documentation for the users : tutorials, function docStrings and reStructuredText documentation etc. The reStructuredText documentation resides inside the `doc` folder in the source code repository.

For building the documentation, you would require `make` and `sphinx`. You can edit the documentation using any text editor, and then generate the HTML output using `make html`. The generated files will reside in `_build/html` folder. Please follow the same conventions for pushing the new documentation as pushing the new code.

3.2 Developer Documentation

The developer documentation (the current document) is generated using LaTeX. We highly recommend using the SnapVX Mailing List to discuss any changes which will improve the quality of this document. Also feel free to post any issues and suggestions to the repository issues page at `Github`.

4 ADMM Details

To solve via ADMM, we introduce a copy of x_j, called z_{jk}, at every edge jk. Note that the same edge also has a z_{kj}, a copy of x_k. We rewrite the original formulation as its equivalent problem,

$$\begin{array}{ll} \text{minimize} & \sum_{i\in\mathcal{V}} f_i(x_i) + \sum_{(j,k)\in\mathcal{E}} g_{jk}(z_{jk}, z_{kj}) \\ \text{subject to} & x_j = z_{jk}, \quad j = 1,\ldots,m, \quad k \in N(j), \end{array}$$

where $N(j)$ is the set of neighbors of node j. Deriving the problem's augmented Lagrangian (see Boyd et al., 2011), we get

$$\begin{aligned} L_\rho(x,z,u) = &\sum_{i\in\mathcal{V}} f_i(x_i) + \sum_{(j,k)\in\mathcal{E}} \Big(g_{jk}(z_{jk}, z_{kj}) - (\rho/2)\left(\|u_{jk}\|_2^2 + \|u_{kj}\|_2^2\right) + \\ &(\rho/2)\left(\|x_j - z_{jk} + u_{jk}\|_2^2 + \|x_k - z_{kj} + u_{kj}\|_2^2\right)\Big), \end{aligned}$$

where u is the scaled dual variable at each edge and $\rho > 0$ is a scalar penalty parameter that determines the tradeoff between primal and dual convergence. ADMM consists of the following steps, with k denoting the iteration number:

$$\begin{aligned} x^{k+1} &= \operatorname*{argmin}_x L_\rho(x, z^k, u^k) \\ z^{k+1} &= \operatorname*{argmin}_z L_\rho(x^{k+1}, z, u^k) \\ u^{k+1} &= u^k + (x^{k+1} - z^{k+1}). \end{aligned}$$

Now, we analyze each of these steps in more detail.

x**-Update.** In the x-update we minimize a separable sum of functions, one per node, so it can be calculated independently at each node and solved in parallel. At node i, this is

$$x_i^{k+1} = \underset{x_i}{\operatorname{argmin}} \left(f_i(x_i) + \sum_{j \in N(i)} (\rho/2) \|x_i - z_{ij}^k + u_{ij}^k\|_2^2 \right).$$

z**-Update.** The z-update is separable across the edges. Note that for edge ij, we need to jointly update z_{ij} and z_{ji}. This becomes

$$z_{ij}^{k+1}, z_{ji}^{k+1} = \underset{z_{ij}, z_{ji}}{\operatorname{argmin}} \Big(g_{ij}(z_{ij}, z_{ji}) + (\rho/2) \left(\|x_i^{k+1} - z_{ij} + u_{ij}^k\|_2^2 + \|x_j^{k+1} - z_{ji} + u_{ji}^k\|_2^2 \right) \Big).$$

u**-Update.** The u-update is also edge-separable. For each variable, this looks like

$$u_{ij}^{k+1} = u_{ij}^k + (x_i^{k+1} - z_{ij}^{k+1}).$$

Global Convergence. Because the problem is convex, ADMM is guaranteed to converge to a global optimum. The stopping criterion can be based on the primal and dual residuals, commonly defined as r and s, being below given threshold values; see Boyd et al., 2011. In this case, these residuals correspond to $r^k = Ax^k - z^k$ and $s^k = \rho A^T(z^{k+1} - z^k)$, where $A \in \mathbb{R}^{2n \times m}$ has $2n$ nonzero elements (all ones), one per row, mapping each z_{ij} to the node where it has an equality constraint. This allows us to stop when x^k and z^k are close, and when x^k (or z^k) does not change much in one iteration. As is typical for ADMM, the algorithm tends to attain modest accuracy relatively quickly, and high accuracy (which in many applications is not needed) only slowly.

Algorithm 1 ADMM Steps

given ϵ^{pri}, ϵ^{dual}
repeat

$$x_i^{k+1} = \underset{x_i}{\operatorname{argmin}} \left(f_i(x_i) + \sum_{j \in N(i)} (\rho/2) \|x_i - z_{ij}^k + u_{ij}^k\|_2^2 \right)$$

$$z_{ij}^{k+1}, z_{ji}^{k+1} = \underset{z_{ij}, z_{ji}}{\operatorname{argmin}} \Big(g_{ij}(z_{ij}, z_{ji}) + (\rho/2) \left(\|x_i^{k+1} - z_{ij} + u_{ij}^k\|_2^2 + \|x_j^{k+1} - z_{ji} + u_{ji}^k\|_2^2 \right) \Big)$$

$$u_{ij}^{k+1} = u_{ij}^k + (x_i^{k+1} - z_{ij}^{k+1})$$

until $\|r^k\|_2 \leq \epsilon^{\text{pri}}$; $\|s^k\|_2 \leq \epsilon^{\text{dual}}$.