Lagrangian-Based Algorithms in SSCNO

Aleksandar Donev

July 2000

1 Module Dual_Newton_SSCNO

The module Dual_Newton_SSCNO contains the top-level numerical optimization routines for strictly convex network optimization. Here I have extracted code from it in order to illustrate the two algorithms which have been implemented as of now. I will give both applied mathematics- and a physics-terminology explanations, with the applied-math version being in parenthesis:

**SSCNO.DTN** The **Truncated Dual Newton Algorithm** (TDN) is based on a dual (Fechel) form of the Lagrangian and modeled after the work of Fukushima et al. It works in potential (multiplier) space only, and calculates the flows (unknowns) as a side-product. It requires coding the arc voltage-current characteristics (elemental arc cost functions) in terms of voltages (tensions), that is, it requires coding $I(V) \left( x^*_{i,j} = x^*_{i,j}(t) = f'_{i,j}^{-1}(t) \right)$ and $R(V) \left( f''_{i,j}(x^*_{i,j}) \right)$ explicitly.

**SSCNO.TSQP** The **Truncated Sequential Quadratic Programming Algorithm** (TSQP) is based on successive quadratic approximations to the Lagrangian and the work of Kliuczewicz and others. It works in flow (primal variable) space, and calculates the potentials (multipliers) as a side-product by solving a random-resistor network subproblem (it uses a dual range-space approach to solving the quadratic subproblems). It requires coding the arc voltage-current characteristics (elemental arc cost functions) in terms of currents (primal variables), that is, it requires coding $V(I) \left( t_{i,j} = t_{i,j}(x) = f'_{i,j}(x_{i,j}) \right)$ and $V(R) \left( f''_{i,j}(x_{i,j}) \right)$ explicitly.

The main part in both of these is solving a random-resistor network (a quadratic programming subproblem) at each iteration, and in both cases an approach based on the conductance matrix (a dual range-space method) is used for this purpose. Only the way in which the conductances and excess flows of the subproblem are assigned and the resulting excess potentials are used changes. I assume that there are other physically-intuitive algorithms similar to these. These two are guaranteed convergence and have been extensively studied in the applied-mathematics literature, which is why I chose them. To repeat, TDN is potential-based and TSQP is flow-based, so I have covered the two main categories of algorithms.
1.1 The Routine \textit{SolveDualSSCNO\_TCGN}

This routine implements both TDN and TSQP together (both of these are \textbf{Truncated (Conjugate Gradient) Newton} (TCGN) methods, but conjugate gradient does not have to be used since direct solvers are also supported.

For TDN, we are given an initial guess for the potentials at the nodes in \textit{nodes\_potentials} (the Lagrange multiplier estimates $\lambda$). For TSQP, we are given an initial guess for the flow through the arcs in \textit{arcs\_flows} (the primal variable $x$). We start an iterative procedure for improving upon these estimates of the actual solution (the solution pair $(\hat{x}, \hat{\lambda})$) which satisfies the two Kirchoff laws (the first two KKT optimality conditions), the flow conservation at the nodes ($A\hat{x} = b$) and the current-voltage characteristics at all of the arcs ($A^T\hat{\lambda} = t = f'(\hat{x})$). To do this we are going to approximate our non-linear network with a random-resistor network (make a quadratic approximation of the Lagrangian) and solve this RRN problem using a conductance-matrix approach (a range-space approach). For this to make sense, at each iteration we must:

- Decide how to assign the conductances/resistances of the arcs of the RRN (make an approximation of the Hessian in the form of a diagonal matrix $C$), in the array \textit{arcs\_conductances}
- Assign supplies-demands for the nodes of the RRN (calculate the gradient of the Lagrangian), which in this sense are really excess flows at the nodes (deviation from KKT conditions) and are in the array \textit{nodes\_excess\_flows}

The two algorithms are going to differ in how they calculate these. The solution of the RRN problem in both cases will produce a solution for the excess potentials at the nodes (a search direction for the multipliers $\Delta \lambda$), which will be stored in \textit{nodes\_excess\_potentials}, but how this solution is used will of course differ.

Here is the main structure of the iterative loop in this routine:

\begin{verbatim}
\langle IterativeLoop 1.1.0.1 \rangle \equiv

iteration = 0  // Iteration counter
TCGN: do  // Start Truncated Newton major iterations
  iteration = iteration + 1
  \langle PrintStatistics 0 \rangle  // Print some statistics and messages (not important)
  \langle CheckConvergence 0 \rangle  // Exit if algorithm converged
  \langle DerivativesCalculation 1.1.1 \rangle  // Calculate the gradient $G$ and the Hessian $H$
  \langle NewtonSolver 1.1.2 \rangle  // Solve Newton's system $H\delta \tau = -G$
  \langle LineSearch 1.1.3 \rangle  // Perform line search along $\delta \tau$
  \end do  // TCGN
\end{verbatim}

2
1.1.1 **Step 1: Gradient/Hessian calculation**

First, we calculate the current excess flow at the nodes ($\Delta b$). This will also assign values to the conductances of the arcs (an approximation of the quadratic subproblem Hessian $C$). Here comes the main difference between the two algorithms.

TDN uses the current voltages across the arcs ($t = A^T \lambda$) to calculate the currents through the arcs using the user-supplied voltage-current characteristic (elemental cost function), that is, it uses $I = I(V) \left( \frac{x(i,j)}{f_{i,j}^0(t)} \right)$. This means that the voltage-current characteristic, i.e., the second Kirchhoff law, is always explicitly satisfied in TDN at each step. So the norm of the excess flow at the nodes, $Ax - b$, is the measure of convergence in this algorithm. The important thing to get is that this algorithm obtains $\text{arcs}_\text{flows}(x)$ only a side-product.

On the other hand, SQP uses the current estimate of the flow through the arcs ($x$), to calculate the voltages across the arcs, that is, it uses $V = V(I) \left( \frac{t(i,j)}{f_{i,j}^0(x(i,j))} \right)$. Now, these voltages do not uniquely determine a set of potentials at the nodes. Therefore, there will be a mismatch in the arcs voltages $\delta t = A \lambda - t$, which will vanish only at the solution. Therefore, we will use $\delta t$ as the measure of convergence of this algorithm. The important thing to get is that this algorithm obtains $\text{nodes}_\text{potentials}(\lambda)$ only a side-product.

In both algorithms, the conductance of the arcs is taken to be the slope of the voltage-current curve at the given point, either $R = V'(I)$ or $R = \frac{1}{f_{i,j}^0(x(i,j))}$. This is all done in one call to a user-supplied elemental cost function.

```plaintext
DerivativesCalculation 1.1.1 ≡

LagrangianGradient: if (_method ≡ SS_CNO_TDN) then  // Use dual Lagrangian
    // Calculate arcs_voltages from nodes_potentials:
    CALL ArcsVoltages(heads_tails = heads_tails, node_offset = n_special_nodes, 
                      nodes_potentials = _nodes_potentials, arcs_voltages = _arcs_voltages) // $t = A^T \lambda$
    // Calculate arcs_flows and arcs_conductances from arcs_voltages:
    CALL DualCostDerivatives(ElementalCosts = ElementalCosts, cost_function = cost_function, 
                               lagrangian_function = dual_lagrangian, arc_offset = n_special_arcs, 
                               arcs_voltages = _arcs_voltages, arcs_flows = _arcs_flows, 
                               arcs_conductances = _arcs_conductances, warm_starts = _warm_starts, 
                               conductances_interval = ([low_conductance, high_conductance]))
    // $x = (f')^{-1}(t)$ and $C = \frac{1}{f(x)}$
    // Calculate nodes_excess_flows ($\Delta b = Ax - b$) from arcs_flows:
    CALL NodesExcessFlows(heads_tails = heads_tails, node_offset = n_special_nodes, 
                          arcs_flows = _arcs_flows, excess_flows = _nodes_excess_flows)  // Store $\Delta b \leftarrow Ax$
    CALL VectorSubtraction(from = supplies_demands, what = _nodes_excess_flows, reverse = T)  // $\Delta b \leftarrow \Delta b - b = Ax - b$
    // In TDN the magnitude of the excess flows is the main measure of convergence:
    _flows_infeasibility_norm = sqrt(dot_product(_nodes_excess_flows, _nodes_excess_flows))
    (ReCheckConvergence_DTN 0)
    ELSE  // Use SQP Lagrangian
```
// Calculate arcs\_voltages and arcs\_conductances from arcs\_flows:
CALL DualCostDerivatives (ElementalCosts = ElementalCosts, cost\_function = cost\_function,
lagrangian\_function = primal\_lagrangian, arc\_offset = n\_special\_arcs,
arcs\_voltages = arcs\_v
tages, arcs\_flows = arcs\_flows,
arcs\_conductances = arcs\_conductances, warm\_starts = warm\_starts,
conductances\_interval = (/ low\_conductance, high\_conductance /))
// \( t = f'(x) \) and \( C = \frac{1}{f'(x)} \)

// Now we assume the network is quadratic and calculate flows through the arcs:
CALL Vector\_Multiplication (first = arcs\_conductances, second = arcs\_v
tages,
product = arcs\_excess\_flows) // \( \Delta x \leftarrow Ct \)
// Now calculate the discrepancy flow with the actual arcs\_flows:
CALL Vector\_Subtraction (from = arcs\_excess\_flows, what = arcs\_flows, reverse = F)
// \( \Delta x \leftarrow \Delta x - x = Ct - x \)

// Calculate the excess flow at the nodes nodes\_excess\_flows from the discrepancy:
CALL Nodes\_Excess\_Flows (heads\_tails = heads\_tails, node\_offset = n\_special\_nodes,
arcs\_flows = arcs\_excess\_flows, excess\_flows = nodes\_excess\_flows)
// \( \Delta b \leftarrow A\Delta x \)
CALL Vector\_Addition (first = nodes\_excess\_flows, second = supplies\_demands)
// \( \Delta b \leftarrow b + \Delta b = (b - Ax) + ACt \)

END IF Lagrangian\_Gradient

This code is used in section 1.1.0.1.
1.1.2 *Step 2: Newton Search direction determination*

In step one we calculated/estimated the excess supplies-demands at the nodes $\text{nodes\_excess\_flows} (b)$, and the estimates of the conductances in $\text{arcs\_conductances} (C)$. Now we are going to approximate our non-linear network with an RRN network (make a quadratic approximation) and solve this RRN network (quadratic subproblem). This will give us the excess/discrepancy potentials at the nodes $\text{nodes\_excess\_potentials} (\Delta \lambda)$, and is done here by solving the Laplacian system (a range-space method):

$$\mathcal{L} \Delta \lambda = (AC^T) \Delta \lambda = b$$

SSCNO can either use a direct Cholesky factorization $\mathcal{L} = L^T L$ or an iterative conjugate gradient to solve this Laplacian (Newton) linear system, whichever the user chooses. Here I must say I cheated a little bit because in the SQP approach we formed the excess flows ($b$) in such a way that the true meaning of the excess potentials ($\Delta \lambda$) is that they are really the new estimates of the potentials ($\lambda$), rather then the correction (update) to the potentials. I still keep the $\Delta$ here though so both algorithms look as close as possible!

\texttt{(NewtonSolver 1.1.2) \equiv}

\texttt{UsePCG: if (\_solution\_method \equiv dual\_network\_pcg) then // Use conjugate gradient iterative solver}

\hspace{1cm}$$\text{nodes\_excess\_potentials} = 0.0,_{\text{wp}}$$ // The initial guess that is suitable with truncation

\hspace{1cm}\texttt{call SolveDualSystem\_PCG(dual\_system = dual\_system)} // Find a descent Newton Direction

\texttt{else} // Use Cholesky factorization

\hspace{1cm}\texttt{call SolveDualSystem\_LLt(dual\_system = dual\_system)} // Find a descent Newton Direction

\texttt{end if UsePCG}

\hspace{1cm}$$\text{It is useful to keep the mean of the underdetermined potentials vector at 0}$$

\hspace{1cm}\texttt{call VectorShift\_vector = _nodes\_excess\_potentials, shift = -SUM(_nodes\_excess\_potentials) / REAL(n\_nodes + n\_special\_nodes, r\_wp))}

This code is used in section 1.1.0.1.

1.1.3 *Step 3: Line search*

Finally, we modify the current estimates of the nodes potentials or arcs flows, hopefully to get them closer to optimal, via a line search. In TDN, this search is conducted in potential (multiplier) space, i.e. one looks for a step size $\alpha$ that minimizes the Lagrangian along the search direction:

$$\min_\alpha \mathcal{L}_{TDN}(\lambda + \alpha \Delta \lambda)$$

This is done in another file and I will not discuss it here. The physical intuition is that the line search will try to modify the potential estimates in the search direction determined from solving the RRN so as to reduce the excess flows at the nodes. Since the cost function is coded in terms of arcs, it is easier to do the above search along the excess arcs voltages, i.e. along $A^T (\lambda + \alpha \Delta \lambda) = t + \alpha \Delta t$. 

5
In SQP, the search is conducted in flow (primal variable) space. To do this one first needs to use the excess potentials ($\Delta \lambda$) to determine the excess arc flows ($\Delta x$), as detailed below, and then conduct a search for a step-size:

$$\min_{\alpha} \mathcal{L}_{SQP}(x + \alpha \Delta x)$$

In both approaches, one the best step size $\alpha$ has been found, either the nodes potentials or the arcs flows are updated accordingly, $\lambda \leftarrow \lambda + \alpha \Delta \lambda$, or $x \leftarrow x + \alpha \Delta x$, which gives us new estimates of these so we are ready to start a new iteration!

(LineSearch 1.1.3) \equiv

LineSearchDirection: \textbf{IF} (_method \equiv \text{SSCNO_TDN}) \textbf{THEN}

\begin{itemize}
  \item \textbf{CALL} ArcsVoltages (\texttt{heads\_tails = heads\_tails, node\_offset = n\_special\_nodes, nodes\_potentials = nodes\_excess\_potentials, arcs\_voltages = arcs\_excess\_voltages})
  \textit{// $\Delta t = A^T \Delta \lambda$}
  \item \textbf{CALL} PerformDualSearch_Root (\texttt{line\_search = line\_search, ElementalCosts = ElementalCosts, initialize = F}) \textit{// $\lambda \leftarrow \lambda + \alpha \Delta \lambda$}
\end{itemize}

\textbf{ELSE}

\begin{itemize}
  \item \textbf{CALL} VectorCopy (\texttt{source = nodes\_excess\_potentials, target = nodes\_potentials}) \textit{// $\lambda = \Delta \lambda$}
  \item \textbf{CALL} ArcsVoltages (\texttt{heads\_tails = heads\_tails, node\_offset = n\_special\_nodes, nodes\_potentials = nodes\_excess\_potentials, arcs\_voltages = arcs\_excess\_voltages})
  \textit{// $\Delta t \leftarrow A^T \Delta \lambda$}
  \item \textbf{CALL} VectorSubtraction (\texttt{from = arcs\_excess\_voltages, what = arcs\_voltages, reverse = F})
  \textit{// $\Delta t = A^T \lambda - t$}
  \item \textbf{CALL} ReCheckConvergence_SQP 0
  \textit{// In SQP the magnitude of the excess voltages is the main measure of convergence:}
  \texttt{v\_infeasibility\_norm = \text{SQRT(D\_PRODUCT(\_arcs\_excess\_voltages, \_arcs\_excess\_voltages))}}
\end{itemize}

\textbf{END IF} LineSearchDirection

This code is used in section 1.1.0.1.