Cost Functions, Derivatives and Inverses
for Power-Law Cost Functions

Aleksandar Donev

July 2001

1 Module Power_Cost_Functions

This documentation is out of date!

This module contains prototype functions for evaluating the cost functions on the arcs and their derivatives. The particular cost function at hand is the power-law cost function:

\[ f_i(x) = C_i |x_i|^\alpha \]

but the interface, purpose and operation of the function PowerElementalCosts should be used as a guide in constructing such a function for different cost functions.

Assume that a flow \( x_i \) flows through some arc with number \( i \). Then the cost on this arc, the elemental cost \( f_i(x_i) \) depends on \( x_i \) and possibly on some other parameters stored in \( \text{arcs~cost~parameters}_i \). A fundamental quantity for dual algorithms is the voltage drop (i.e. voltage) across the arc, \( t_i = f'_i(x_i) \), which is in fact the gradient of the cost function. Another quantity is the resistance \( r_i = f''_i(x) \), which is in fact the Hessian of the primal-cost function.

For strictly convex elemental costs \( f_i(x) \) though, as required by this optimization library, the voltage should be invertible, i.e. either specifying the voltage or the flow uniquely determines all quantities via \( x_i = f_i^{-1}(t_i) \). Therefore, out of the four quantities \( x_i, t_i, r_i \) and \( f_i \) either one of the first two can be used as input, and some or all of the remaining quantities required to be evaluated.

Also notice that some of these may be easier to calculate if the flow is known, some if the voltage is known, and yet some if both are known. For example, the power cost \( f_i = |x|^\alpha \) requires expensive real exponentiation when only the flow is given. But if both the flow and voltage are known (from a previous calculation), then the cost is \( f_i = \alpha^{-1} x_i t_i \), which is very cheap to calculate. Same can be done for the resistance.

Finally, yet another important possibility for this library to incorporate is giving approximate values (guesses) for some of these quantities. These can possibly be used as starting points in iterative refinement methods inside the routine. The most famous is the power-law cost, for which Newton’s method is often used. In this library, the estimates of the voltages or flows (from previous iterations) can be used as initial guesses here.

Therefore, the input argument 4-letter string arguments\_status tells for each of the input variables flow, voltage, resistance and cost (in this order) whether the variable is:
'F' if the value is known exactly and is therefore fixed inside the routine

'W' if the value needs to be calculated and the current value is a guess (warm start) for the exact result

'C' if the value is needed but no guess is provided (cold start) and the input value is this meaningless

'H' if the value is not needed and the current value is an approximate value (helpful hint), as in 'W'

'D' for dummy—the value should be left untouched and ignored by the routine.

The library will never rely on any default values (i.e., only the above 6 letters are valid—always in uppercase as well!). At least the flow or the voltage must be 'F' (known exactly), and other than that there is no restriction. I decided to implement this interface in such an entangled way to provide enough flexibility to the user for most cases that occur in practice, and yet bundle it all into one easy-to-call routine that can be passed as a dummy argument to the optimization libraries and called from inside them with ease.

At first I wanted to make these vector routines, i.e. one call for all arcs, but, since in reality it is expected that these functions will be rather costly (transcendental in the very least), I decided to do a CALL for each arc (passed here via the argument arc_index). If the cost functions are very fast, such as quadratic functions, then you are mostly stuck with the subroutine calling overhead. Shouldn’t be much of a penalty (I hope!).

Finally, the argument tolerance gives the minimal precision with which the variables here need to be calculated. Usually this would be of the order of the machine precision, but may be higher if only approximate values are needed.

"WEAVE.f90" 1.0.1 ≡

MODULE Power_Cost_Functions
  USE Precision
  USE Error_Handling
  USE System_Monitors
  USE Network_Data_Structures
  USE Network_Data_Types
  USE Power_Cost_Parameters
  USE Network_Data_Types
  IMPLICIT NONE
  PUBLIC :: PowerElementalCosts
PRIVATE
  INTEGER, SAVE, PUBLIC :: power_elemental_costs_timer = -1
CONTAINS
  (PowerElementalCosts 1.1.1)
END MODULE Power_Cost_Functions

1.1 Cost Function and Related Derivatives Calculation
This is a an actual implementation of the routine ElementalCosts for the power-law cost function \( f(x) = C|x|^\alpha \). For this cost-function we have that \( f'(x) = t = C\alpha \text{sign}(x)|x|^{\alpha-1} \), which can be inverted by taking a root (i.e. an exponentiation). Exponentiations can be saved if both the flow and the voltage are known. I have tried to avoid all numerical underflows here and various other exceptions by using a small "correction" \( \text{eps} \) in the tricky formulae, but for large values of \( \alpha \) there is little one can do about the numerical stability here.

\[
\langle \text{PowerElementalCosts 1.1.1} \rangle \equiv \\
\text{SUBROUTINE PowerElementalCosts} (\text{cost\_function}, \text{arguments\_status}, \text{tolerance}, \text{arcs\_indices}, \\
\text{arcs\_flows}, \text{arcs\_voltages}, \text{arcs\_resistances}, \text{arcs\_costs}) \\
\text{USE Precision} \\
\text{USE Network\_Data\_Types} \\
\_TYPE (\text{Network\_SC\_Cost}), \text{INTENT} (\text{inout}) :: \text{cost\_function} \quad / / \text{CLASS in F2x} \\
\text{CHARACTER (LEN = 4)}, \text{INTENT} (\text{in}) :: \text{arguments\_status} \quad / / \text{For example "FOOD"} \\
\text{REAL (kind = r\_wp)}, \text{INTENT} (\text{in}) :: \text{tolerance} \\
\text{INTEGER (kind = i\_wp)}, \text{DIMENSION (2)}, \text{INTENT (in)} :: \text{arcs\_indices} \\
\text{REAL (kind = r\_wp)}, \text{DIMENSION (arcs\_indices1 :), INTENT (inout), OPTIONAL :: arcs\_flows, \\
arcs\_voltages, arcs\_resistances, arcs\_costs} \\
\text{LOGICAL :: known\_flow, known\_voltage} \quad / / \text{Indicators—I do not care about warm starts here} \\
\text{CHARACTER :: status} \quad / / \text{A temporary} \\
\text{REAL (kind = r\_wp)} :: \text{eps} \quad / / \text{A small number used to correct underflows} \\
\nosmall \\
\text{// The following will be handled via exceptions in F2002:} \\
\text{eps = MAX(EPSILON(1.0, r\_wp), tolerance)} \quad / / \text{To avoid divisions by zero} \\
\text{CALL Start Timer (power\_elemental\_costs\_timer)} \quad / / \text{Internal timer} \\
\text{PowerElementalCost(arcs\_flows, arcs\_voltages, arcs\_resistances, arcs\_costs,} \\
\text{arcs\_cost\_parameters1, arcs\_indices1(1):arcs\_indices1(2))} \\
\text{// Call the function macro with array arguments} \\
\text{CALL Stop Timer (power\_elemental\_costs\_timer)} \quad / / \text{Internal timer} \\
\text{END SUBROUTINE PowerElementalCosts} \\
\]

This code is used in section 1.0.1.

"WAVE.f90" 1.1.2 \equiv \\
@m _PowerElementalCost(flow, voltage, resistance, cost, cost\_coefficient) \\
\nosmall \\
\text{// Flow calculation: */} \\
\text{status = arguments\_status}; \\
\text{known\_flow = (status \equiv 'F')} \\
\text{IF ((NOT known\_flow) \& (status \neq 'D')) THEN} \quad / / \text{Calculate the flow from the voltage} \\
\text{flow = SIGN(1.0, wp), voltage) \* (ABS(voltage) / (alpha * cost\_coefficient + eps))^{inv\_alpha\_minus_1} \\
\text{// x = sign(t) \* \left[ \frac{1}{\alpha - 1} \right]^{\frac{1}{\alpha - 1}}} \\
\text{known\_flow = T} \\
\text{END IF} \\
\nosmall \\
\text{/* Potential calculation */}
\[
\text{status} = \text{arguments.status}_{2:2}
\]
\[
\text{known\_voltage} = (\text{status} \equiv 'F')
\]
\[
\text{IF} \left( \lnot\text{known\_voltage} \land (\text{status} \neq 'D') \right) \text{ THEN} \quad \text{// Calculate the voltage from the flow}
\]
\[
voltage = \alpha \ast \text{cost\_coefficient} \ast \text{SIGN}(1.0_{\text{wp}}, \text{flow}) \ast ((\text{ABS}((\text{flow}) + \epsilon)\alpha^{-1.0_{\text{wp}}})
\]
\[
\text{// } t = C \alpha \text{ sign}(x) |x|^{q-1}
\]
\[
\text{known\_voltage} = T
\]
\[
\text{END IF}
\]
\[
/* \text{ Resistance calculation */}
\]
\[
\text{status} = \text{arguments.status}_{3:3}
\]
\[
\text{IF} \left( (\text{status} \neq 'F') \land (\text{status} \neq 'D') \right) \text{ THEN} \quad \text{// Calculate the resistance from the flow or the voltage}
\]
\[
\text{IF} \left( \text{known\_voltage} \land \text{known\_flow} \right) \text{ THEN} \quad \text{// Best choice—no exponentiation}
\]
\[
\text{resistance} = (\alpha - 1.0_{\text{wp}}) \ast \text{ABS}(\text{voltage}) / (\text{ABS}((\text{flow}) + \epsilon))
\]
\[
\text{ELSE IF} \left( \text{arguments\_status}_{1:1} \equiv 'F' \right) \text{ THEN} \quad \text{// Only the flow is known}
\]
\[
\text{resistance} = \alpha \ast (\alpha - 1.0_{\text{wp}}) \ast \text{cost\_coefficient} \ast ((\text{ABS}((\text{flow}) + \epsilon)\alpha^{-2.0_{\text{wp}}})
\]
\[
\text{ELSE IF} \left( \text{arguments\_status}_{2:2} \equiv 'F' \right) \text{ THEN} \quad \text{// Only the voltage is known}
\]
\[
\text{resistance} = (\alpha - 1.0_{\text{wp}}) \ast \text{ABS}(\text{voltage}) \ast (\text{ABS}(\text{voltage}) / (\alpha \ast \text{cost\_coefficient} + \epsilon) + \epsilon^{-\text{inv\_alpha\_minus\_I}})
\]
\[
\text{ELSE} \quad \text{// Should never occur!!!}
\]
\[
\text{CALL CriticalError("Neither flow nor potential known", "PowerElementalCosts")}
\]
\[
\text{resistance} = 0.0_{\text{wp}}
\]
\[
\text{END IF}
\]
\[
\text{END IF}
\]
\[
/* \text{ Cost calculation */}
\]
\[
\text{status} = \text{arguments.status}_{4:4}
\]
\[
\text{IF} \left( (\text{status} \neq 'F') \land (\text{status} \neq 'D') \right) \text{ THEN} \quad \text{// Calculate the cost from the flow or the voltage}
\]
\[
\text{IF} \left( \text{known\_voltage} \land \text{known\_flow} \right) \text{ THEN} \quad \text{// Best choice—no exponentiation}
\]
\[
\text{cost} = (\text{flow} \ast \text{cost\_coefficient}) / \alpha \quad \text{// No need for absolute values here}
\]
\[
\text{ELSE IF} \left( \text{arguments\_status}_{1:1} \equiv 'F' \right) \text{ THEN} \quad \text{// Only the flow is known}
\]
\[
\text{cost} = \text{cost\_coefficient} \ast \text{ABS}(\text{flow}) \ast \alpha
\]
\[
\text{ELSE IF} \left( \text{arguments\_status}_{2:2} \equiv 'F' \right) \text{ THEN} \quad \text{// Only the voltage is known}
\]
\[
\text{cost} = \text{cost\_coefficient} \ast (\text{ABS}(\text{voltage}) / (\alpha \ast \text{cost\_coefficient} + \epsilon) + \epsilon) \ast \text{inv\_alpha\_minus\_I}
\]
\[
\text{ELSE} \quad \text{// Should never occur!!!}
\]
\[
\text{CALL CriticalError("Neither flow nor potential known", "PowerElementalCosts")}
\]
\[
\text{cost} = 0.0_{\text{wp}}
\]
\[
\text{END IF}
\]
\[
\text{END IF}
\]
```plaintext
@m CASE_TYPE TYPE
@m _TYPE TYPE
@m NULL > NULL()
@m _PRIVATE PRIVATE
@m _SIZE(array, _kind,...)
 @IFELSE (#0, 0, INT(SIZE(array), KIND=_kind), INT(SIZE(array,#), KIND=_kind))
@m _MAXLOC(array, _kind,...)
 @IFELSE (#0, 0, INT(MAXLOC(array), KIND=_kind), INT(MAXLOC(array,#), KIND=_kind))
@m _MINLOC(array, _kind,...)
 @IFELSE (#0, 0, INT(MINLOC(array), KIND=_kind), INT(MINLOC(array,#), KIND=_kind))
@m _LBOUND(array, _kind,...) @IFELSE (#0, 0, INT(LBOUND(array, DIM=1), KIND=_kind),
INT(LBOUND(array, #), KIND=_kind))
@m _UBOUND(array, _kind,...) @IFELSE (#0, 0, INT(UBOUND(array, DIM=1), KIND=_kind),
INT(UBOUND(array, #), KIND=_kind))
@m _GenericInterface(generic_name,...)
   INTERFACE generic_name MODULE PROCEDURE #.
   END Declare INTERFACE generic_name
@m _Declare_i_word(...)
   INTEGER :: #.
@m _Declare_i_wp(...)
   INTEGER (KIND = i_wp) :: #.
@m _Declare_r_wp(...)
   REAL (KIND = r_wp) :: #.
@m _Declare_r_sp(...)
   REAL (KIND = r_sp) :: #.
@m _Declare_r_dp(...)
   REAL (KIND = r_dp) :: #.
@m _FullExtent(_rank) : DO (DIM, 2, _rank) { : }
@m _VarSequence(_variable, _start, _end)
   _variable##_start DO (DIM, $\text{SEVAL}(\_start + 1), _end) \{ , _variable@&DIM \}
@m _NestedLoopStart(_variable, _array, _rank, _kind)
   DO (_DIM, _rank, 1, -1) \{ DO _variable@&DIM = LBOUND(_array, _kind, DIM = _DIM),
   UBOUND(_array, _kind, DIM = _DIM) \} 
@m _NestedLoopEnd(_rank) DO (_DIM, 1, _rank) \{ END DO \}
@m _Dummy(...)
@m _DisplayArray(message, array)
   IF (SIZE(array) \leq 20) THEN
      WRITE(message, print_unit, "(A)”) message
      WRITE(message, print_unit, "(2005.2)”) array
   END IF
```

1