Submitted to *INFORMS Journal on Computing*
manuscript (Please, provide the mansucript number!)

Online supplement for "Numerically safe lower bounds for the Capacitated Vehicle Routing Problem"

Ricardo Fukasawa, Laurent Poirrier
Department of Combinatorics and Optimization, Faculty of Mathematics, University of Waterloo,
Waterloo, ON, N2L3G1, Canada
{rfukasawa,lpoirrier}@uwaterloo.ca

---

1. Implementation details for setting rounding mode

Technically, the workaround developed in Section 4.3 amounts to introducing a *compiler memory barrier* between the call to `fesetround()` and the subsequent arithmetic operations to enforce a specific ordering. In C, function calls act as compiler memory barriers as long as the compiler cannot see the function definition. They present two advantages over manually inserting a memory barrier (e.g. with `asm volatile("" : : : "memory");` in GCC). Firstly, function calls are portable and specified by the standard. Secondly, memory barriers have no impact on local variables (even if they are stored on the stack) unless a pointer to them is taken previously in the execution flow. Implementing arithmetic operations in a separate function ensures that no local variables could escape our memory ordering constraints.

Several alternatives to a function call are possible.

One may create a no-operation function `touch(double *)` defined outside of the compilation unit, to which we pass pointers to the local variables involved. Equivalently, we can

create a wrapper `fesetround_wrapper()` for `fesetround()` that takes a variable number of arguments, to which we pass those same pointers.

```
void touch(double *v)
{
}
```

```
int fesetround_wrapper(int mode, ...)
{
      return(fesetround(mode));
}
```

In both cases, the function does not modify its arguments, but this method forces the compiler to store them to memory. Since the call to `fesetround()` acts as a compiler memory barrier, it produces the desired outcome. In the case of the wrapper function, the overhead could be limited to the storage of local variables to memory by inlining the code of `fesetround()` in the wrapper function. Examples follow.

```
fesetround(FE_DOWNWARD);
touch(&a)
x = 1.0 / a;
touch(&x)
fesetround(FE_TONEAREST);
touch(&a)
y = 1.0 / a;
```

```
fesetround_wrapper(FE_DOWNWARD, &a);
x = 1.0 / a;
fesetround_wrapper(FE_TONEAREST, &x, &a);
y = 1.0 / a;
```

A second possibility is to use an identity function `self(double)` which returns its argument. In this case, local variables may not need to be stored to memory if the ABI permits argument passing through registers (as is the case in x86_64), but the function calls to `self()` cannot be avoided.

```
fesetround(FE_DOWNWARD);
x = self(1.0 / self(a));
fesetround(FE_TONEAREST);
y = self(1.0 / self(a));
```

Finally we can create a `safe_double` class and define all its operator methods in a separate compilation unit. This approach has most overhead (constructors and destructors will be called as well as operator methods), but it requires less caution from its user.

```
safe_double b = a;
fesetround(FE_DOWNWARD);
safe_double x = 1.0 / b;
fesetround(FE_TONEAREST);
safe_double y = 1.0 / b;
```

It is not perfect however, as care must be taken with implicit casts. In the following example indeed,

```
fesetround(FE_DOWNWARD);
safe_double x = 1.0 / (double)a;
```

the division may be performed before the call to `fesetround()`, since only the result of `1.0 / (double)a` is cast to `safe_double`.