Operation count of a Measurement Equation

M.A. Brentjens

November 11, 2005

Contents

1 Introduction **3**

2 Notation **3**

3 Measurement Equation **4**

4 Solving a timeslot **4**
4.1 Model patch visibilities 4
4.2 At the start of a solver iteration 5
4.3 Derivative evaluation 5

5 Computational costs **5**
5.1 Elementary operations 5
5.2 Partial Fourier kernel k_{ipsrt} 6
5.3 UVW coordinates 6
5.4 Computing a patch visibility 6
5.5 Corrupting a patch 7
5.6 Adding all patches 7

6 High level cost equation **7**

7 Software **8**
7.1 Coding convention 8
7.2 `Hardware` class 8
7.3 `InstructionStats` class 8
7.4 Helper functions 8

8 Examples **9**
8.1 WSRT observation 3C 343 10
8.2 Full LOFAR, 10 kHz channels 11
8.3 Full LOFAR, 200 channels 12
8.4 Full LOFAR, 10 patches 13
8.5 LOFAR core, 32 stations, 10 seconds, 200 subbands 14
8.6 LOFAR core, 32 stations, 100 patches 15

9 Comparison with MeqTree system **16**

10 A bit more realistic **18**

11 How to reduce cost **19**

12 Conclusions **19**

Table 1: List of symbols

Symbol	Explanation
x	Scalar
z^*	Complex conjugate of z
$\mathbf{x}$	Vector
X	2×2 matrix
$\mathbf{a} \cdot \mathbf{b}$	Vector inner product between **a** and **b**
V_{ijprt}	Uncorrupted patch visibility
J_{ipt}	Station / sky patch Jones matrix
$\mathbf{l}_{ps}$	"lmn" coordinate vector of source s of patch p
$\mathbf{u}_{it}$	"uvw" coordinate of station i at timeslot t
c	Speed of light in vacuum
E_{psr}	Normalized coherency matrix of source s of patch p at frequency channel r
$\tilde{\mathrm{V}}_{ijtr}$	Fully corrupted model visibility on baseline between stations i and j
t	timeslot index (0-based)
r	frequency channel index (0-based)
R	number of frequency channels
N	number of stations
B	number of baselines
P	number of patches
F	number of beams with R channels each
S_p	number of sources in patch p
S	total number of sources
ν_r	central frequency of channel r
$\delta\nu$	channel frequency increment

1 Introduction

This memo is very much work in progress and has not yet been verified by anyone besides the author.

ASTRON has been working on implementing a self calibration system for LOFAR since 2001/2002. To my knowledge, no one has ever estimated the minimum amount of operations needed to evaluate a Measurement Equation in a more-or-less realistic scenario. In this memo I analyze the number of operations needed for the evaluation of a particular ME (predict) for the purpose of solving for instrumental or ionospheric corrections.

2 Notation

The notation adhered to in this memo is summarized in Tab. 1.

3 Measurement Equation

At a given timeslot t and frequency channel r, the visibility on a baseline between station i and station j of a collection of patches p and point sources ps is given by

$$\tilde{\mathrm{V}}_{ijrt} = \sum_{p=1}^{P} \mathrm{J}_{ipt} \mathrm{V}_{ijprt} \mathrm{J}^{\dagger}_{jpt}. \tag{1}$$

Equation (1) is a *very* minimal direction dependent measurement equation. The frequency independent Jones matrix J defaults to the unit matrix. The model patch visibility matrix

$$\mathrm{V}_{ijprt} = \sum_{s=1}^{S_p} k_{ipsrt} \mathrm{E}_{psr} k^{*}_{jpsrt}, \tag{2}$$

where the normalized source coherency matrix

$$\mathrm{E}_{psr} = \frac{1}{n_{ps}} \begin{pmatrix} < xx >_{psr} & < xy >_{psr} \\ < yx >_{psr} & < yy >_{psr} \end{pmatrix}. \tag{3}$$

The partial Fourier kernel k_{ipsrt} is a complex scalar given by

$$k_{ipsrt} = \mathrm{e}^{-2\pi \mathrm{i}(u_{it} l_{ps} + v_{it} m_{ps} + w_{it}(n_{ps}-1))\nu_r / c}. \tag{4}$$

4 Solving a timeslot

In this memo we assume that only two coefficients per J_{ipt} are solvable. For example the diagonal phases, diagonal amplitudes, off-diagonal amplitudes, or off-diagonal phases.

When solving for instrumental calibration coefficients, there are three ways of evaluating the measurement equation. First, one needs to compute the model patch visibilities. Then, at the start of each iteration, one should multiply all patch visibilities with the updated sets of Jones matrices. The last phase in every iteration is the computation of the derivatives of all visibilities to the solvable coefficients of the Jones matrices. The costs of adding equations to the solver and computing the matrix solution are not modelled in this memo.

The following sub sections treat these "modes" in more detail.

4.1 Model patch visibilities

If one attempts to solve for phases or amplitudes of J, the model patch visibility, is invariant. It must therefore be computed only once at the beginning of the solution of a timeslot.

If the channels r are equidistant, the partial Fourier kernels k_{ipsrt} can be computed very efficiently:

$$k_{ipsrt} = \mathrm{e}^{-2\pi \mathrm{i}(u_{it} l_{ps} + v_{it} m_{ps} + w_{it}(n_{ps}-1))(\nu_0 + r\delta\nu)/c} \tag{5}$$

$$k_{ipsrt} = k^{\nu_0}_{ipst} \times \left(k^{\delta\nu}_{ipst}\right)^{r}, \tag{6}$$

where

$$k^{\nu_0}_{ipst} = \mathrm{e}^{-2\pi \mathrm{i}(u_{it} l_{ps} + v_{it} m_{ps} + w_{it}(n_{ps}-1))\nu_0 / c} \tag{7}$$

and

$$k_{ipst}^{\delta\nu} = \mathrm{e}^{-2\pi\mathrm{i}(u_{it}l_{ps}+v_{it}m_{ps}+w_{it}(n_{ps}-1))\delta\nu/c}. \tag{8}$$

After this, one needs one complex scalar-scalar multiplication and one complex scalar-matrix multiplication in order to compute the model visibiity of one source. Then one requires $P(S_p-1)$ matrix additions in order to compute V_{ijprt} for all patches. This must be repeated for all B baselines and R channels. It is assumed that all baselines are used, so

$$B = \frac{N(N-1)}{2}. \tag{9}$$

That is nevertheless seldom the case in reality. Usually, the shorter baselines are not used in the calibration because of difficulties in modelling extended sources.

4.2 At the start of a solver iteration

The first step of a solver iteration is a full prediction, given the current set of Jones matrices. Given the invariant V_{ijprt}, one needs to multiply all patches with their current Jones matrices and add them to form a visibility point. This involves two complex matrix-matrix multiplications per patch, and $P-1$ complex matrix additions. This must be repeated for all B baselines and R channels.

4.3 Derivative evaluation

For all visibility points, one needs to evaluate the derivative of the measurement equation with respect to all solvable parameters. We approximate derivatives by forward differencing. That means that for each derivative, one needs a partial evaluation of the measurement equation. Because each coefficient is specific to one patch only, one must multiply this particular patch with the affected Jones matrices and add all patches to obtain the perturbed visibility. The matrix multiplication must be done for all channels and all $N-1$ baselines involving the perturbed Jones matrix.

5 Computational costs

5.1 Elementary operations

All basic math involves complex numbers. The operations needed for basic algebra are listed in the following table:

Operation	multiply	add
compl. scalar-scalar mul	4	2
compl. scalar-matrix mul	16	8
compl. matrix-matrix mul	32	24
compl. matrix-matrix add	0	8
real matrix-3-vector mul	9	6

5.2 Partial Fourier kernel k_{ipsrt}

Computing l, m, and $n-1$ is done per source and is done once per calibration. Those costs are therefore negligible compared to all other operations. I consider them non existent.

The cost of evaluating k_{ipsrt} for all R channels is:

mul: $7 + 4(R-1) + 1$

add: $3 + 2(R-1)$

sin: 2

cos: 2

The "+1" in the mul is the multiplication by $1/c$.

The partial Fourier kernel must be computed for all station-source combinations. That is, that is, NPS_p times for each timeslot.

5.3 UVW coordinates

A very rough estimate, assuming 6 3×3 rotation matrices (three "nominal" rotations, three correction rotations). One needs to compute, for each matrix, a sine and cosine. Some time calculations (Taylor series?) are also required. Let's estimate that one needs 20 terms, yielding 20 more multiplications and 20 more additions for the date/time computations.

The total cost for one UVW coordinate computation would then be estimated as

mul: 74

add: 56

sin: 6

cos: 6

Because the *uvw* coordinates are computed only once for all stations (N times per timeslot, independent of baseline and frequency) their computing cost is almost irrelevant.

5.4 Computing a patch visibility

Given that E_{psr}, k_{ipsrt}, and k_{jpsrt}, are already available, one needs S_p complex scalar-scalar multiplications and S_p complex scalar-matrix multiplications to predict all source visibilities. Combining these source visibilities into a patch visibility costs an additional $S_p - 1$ complex matrix additions. The cost for one patch visibility is then

mul: $20S_p$

add: $18S_p - 8$

This needs to be done PRB times per timeslot. It is *not* necessary to repeat this step for each iteration as the sky model is held constant.

5.5 Corrupting a patch

Corrupting a patch visibility for one baseline, one channel costs two complex matrix-matrix multiplications.

mul: 64

add: 48

This needs to be done PRB times at the start of each solver iteration.

For each J_{ipt} parameter derivative one has to re-corrupt all patch visibilities that involve this parameter. That means at most $(N-1)R$ times per parameter per timeslot.

If there are two solvable parameters per patch per station in one solution: $2(N-1)P \times BR$ times per timeslot per solver iteration for phases and $2NP \times BR$ for amplitudes.

5.6 Adding all patches

Adding all patches costs $P-1$ complex matrix-matrix additions.

mul: 0

add: $8P-8$

This must be repeated for all baselines and channels at the start of every iteration. That is, BR times. In the derivative calculations, the patches must be added once for each derivative. That means $(2(N-1)P+1)BR$ times in total per timeslot for phase solutions and $(2NP+1)BR$ for amplitudes. Therefore, this component to the total computation time scales with the *square* of the number of patches! Fortunately, it is a relatively cheap operation. It nevertheless is *the* limiting factor when the number of patches is larger than approximately 20.

6 High level cost equation

The total cost for computing the patch visibilities is equal to the cost of the uvw, k_{ipsrt}, source visibilities V_{ijpsrt}, and the combination of all source visibilities into patch visibilities V_{ijprt}.

At the start of each iteration, one must corrupt P patches and combine them all into the unperturbed visibility V_{ijrt}. Then one must corrupt $N-1$ patches and combine them for all $\approx 2NP$ solvables.

A few relevant lines from the `Python` script are

```
insSkyModel = insUVW + insStatSourceDFT + insSourcePatchVis +\
              insAddSources

insFullEveryIteration = insCorruptPatch*cPatches + insAddPatches
insDerivatives        = insCorruptPatch*((cStations-1.0)/cBaselines)\
                                    + insAddPatches


ins = (insSkyModel + (insFullEveryIteration +\
                        (insDerivatives*cSolvablesPerTimeslot)\
                        )*cIterations)*cBeams
```

Currently, the MeqTree solver typically converges in 4 iterations. Because only 2 out of 8 coefficients are solved for in one iteration, one needs to do this 4 times. That is, in the end one needs of order 16 iterations. If there are multiple beams with R channels, but in independent directions, then the entire cost must be multiplied by the number of beams.

7 Software

I have written a small utility in Python that does the actual operations counting. I first introduce the coding convention. Then I proceed to explain which classes exist and how they interact.

7.1 Coding convention

I use *apps Hungarian* (Charles Simonyi, Microsoft). Prefixes used in the script:

c Count of something

s String

ins `InstructionStats` instance

hw `Hardware` instance

cst Cost of an operation in CPU clock cycles

rf Frequency in Hz

7.2 `Hardware` class

The `Hardware` class holds a description of a CPU. It is used to help converting instruction counts to clock cycles and execution times in seconds. The actual conversion is done in `InstructionStats.SFromHw()`

7.3 `InstructionStats` class

The `InstructionStats` class helps with instruction arithmetic. It holds counts for the multiply, add, sine and cosine instructions. One can add two `InstructionStats` instances, or multiply an `InstructionStat` instance with a real number.

The `InstructionStats.SFromHw(hw)` method converts instruction counts to clock cycles, given a `Hardware` instance `hw`.

7.4 Helper functions

There are several "constructor" functions that construct `InstructionStats` instances for useful cases, such as a scalar-scalar product, a matrix-matrix product, a smart, and a stupid implementation of k_{ipsrt}.

8 Examples

This section contains a number of example outputs of the `predict_runtime.py` script. The first table always gives the configuration that was used in the computations. The second table gives the operation in the first column, the number of times the operation is called in the second column, and the number of clock cycles in the third column. For the clock cycle computation, the following CPU model was used.:

name: AMD 1900+

clock: 1.6 GHz

mul: 1 cycle

add: 1 cycle

sin: 20 cycles

cos: 20 cycles

The results that are quoted are the following:

SkyModel: the model patch visibilities V_{ijprt}

CorruptPatches: $\mathrm{J}_{ipt}\mathrm{V}_{ijprt}\mathrm{J}^{\dagger}_{jpt}$

AddPatches: adding all patches to form V_{ijrt}

The amount of CPU seconds needed to compute all predict costs for one timeslot is given below the count table. Finallyi, the minimum amount of TFlops needed to do all ME evaluations and derivative computations in real time is quoted. This number assumes 100% CPU efficiency adn does *not* include adding equations to the solver and performing the solver step itself.

8.1 WSRT observation 3C 343

Settings:

Parameter	Value
Stations	14.0
Channels per beam	16.0
Patches	2.0
Sources per patch	1.0
Sources in total	2.0
Solvables per timeslot	52.0
Solver Iterations	4
Beams	1
Integration time	30.0

Results:

Component	Instructions	Cycles	
SkyModel			
Multiply	61180	61180	
Add	30828	30828	
Cosine	140	2800	
Sine	140	2800	
CorruptPatch			
Multiply	3514367	3514367	
Add	2635775	2635775	
Cosine	0	0	
Sine	0	0	
AddPatches			
Multiply	0	0	
Add	2469376	2469376	
Cosine	0	0	
Sine	0	0	+
Multiply	3575548	3575548	
Add	5135980	5135980	
Cosine	140	2800	
Sine	140	2800	

Time@AMD 1900+: **0.00545 seconds**

8.2 Full LOFAR, 10 kHz channels

Settings:

Parameter	Value
Stations	77.0
Channels per beam	3200.0
Patches	20.0
Sources per patch	500.0
Sources in total	10000.0
Solvables per timeslot	3040.0
Solver Iterations	16
Beams	1
Integration time	1.0

Results:

Component	Instructions	Cycles
SkyModel		
Multiply	1882499085698	1882499085698
Add	1688806662312	1688806662312
Cosine	1540462	30809240
Sine	1540462	30809240
CorruptPatch		
Multiply	948830208000	948830208000
Add	711622656000	711622656000
Cosine	0	0
Sine	0	0
AddPatches		
Multiply	0	0
Add	69247530598400	69247530598400
Cosine	0	0
Sine	0	0
		+
Multiply	2831329293698	2831329293698
Add	71647959916712	71647959916712
Cosine	1540462	30809240
Sine	1540462	30809240

Time@AMD 1900+: **46549.5942681 seconds**
Real-time calibration requires at least **74.5 TFlops**.

8.3 Full LOFAR, 200 channels

Settings:

Parameter	Value
Stations	77.0
Channels per beam	200.0
Patches	20.0
Sources per patch	500.0
Sources in total	10000.0
Solvables per timeslot	3040.0
Solver Iterations	16
Beams	1
Integration time	1.0

Results:

Component	Instructions	Cycles
SkyModel		
Multiply	117659085698	117659085698
Add	105551142312	105551142312
Cosine	1540462	30809240
Sine	1540462	30809240
CorruptPatch		
Multiply	59301888000	59301888000
Add	44476416000	44476416000
Cosine	0	0
Sine	0	0
AddPatches		
Multiply	0	0
Add	4327970662400	4327970662400
Cosine	0	0
Sine	0	0
		+
Multiply	176960973698	176960973698
Add	4477998220712	4477998220712
Cosine	1540462	30809240
Sine	1540462	30809240

Time@AMD 1900+: **2909 seconds**
Real-time calibration requires at least **4.7 TFlops**.

8.4 Full LOFAR, 10 patches

Settings:

Parameter	Value
Stations	77.0
Channels per beam	200.0
Patches	10.0
Sources per patch	1000.0
Sources in total	10000.0
Solvables per timeslot	1520.0
Solver Iterations	16
Beams	1
Integration time	1.0

Results:

Component	Instructions	Cycles	
SkyModel			
Multiply	117659085698	117659085698	
Add	105597958312	105597958312	
Cosine	1540462	30809240	
Sine	1540462	30809240	
CorruptPatch			
Multiply	29650944000	29650944000	
Add	22238208000	22238208000	
Cosine	0	0	
Sine	0	0	
AddPatches			
Multiply	0	0	
Add	1025382758400	1025382758400	
Cosine	0	0	
Sine	0	0	
			+
Multiply	147310029698	147310029698	
Add	1153218924712	1153218924712	
Cosine	1540462	30809240	
Sine	1540462	30809240	

Time@AMD 1900+: **813 seconds**
Real-time calibration requires at least **1.3 TFlops**.

8.5 LOFAR core, 32 stations, 10 seconds, 200 subbands

Settings:

Parameter	Value
Stations	32.0
Channels per beam	200.0
Patches	20.0
Sources per patch	500.0
Sources in total	10000.0
Solvables per timeslot	1240.0
Solver Iterations	16
Beams	24
Integration time	10.0

Results:

Component	Instructions	Cycles	
SkyModel			
Multiply	482334776832	482334776832	
Add	431242795008	431242795008	
Cosine	15364608	307292160	
Sine	15364608	307292160	
CorruptPatch			
Multiply	237699072000	237699072000	
Add	178274304000	178274304000	
Cosine	0	0	
Sine	0	0	
AddPatches			
Multiply	0	0	
Add	7185521049600	7185521049600	
Cosine	0	0	
Sine	0	0	
			+
Multiply	720033848832	720033848832	
Add	7795038148608	7795038148608	
Cosine	15364608	307292160	
Sine	15364608	307292160	

Time@AMD 1900+: **5322 seconds**
Real-time calibration requires at least **0.9 TFlops**.

8.6 LOFAR core, 32 stations, 100 patches

Settings:

Parameter	Value
Stations	32.0
Channels per beam	200.0
Patches	100.0
Sources per patch	100.0
Sources in total	10000.0
Solvables per timeslot	6200.0
Solver Iterations	16
Beams	24
Integration time	10.0

Results:

Component	Instructions	Cycles	
SkyModel			
Multiply	482334776832	482334776832	
Add	429719083008	429719083008	
Cosine	15364608	307292160	
Sine	15364608	307292160	
CorruptPatch			
Multiply	1188495360000	1188495360000	
Add	891371520000	891371520000	
Cosine	0	0	
Sine	0	0	
AddPatches			
Multiply	0	0	
Add	187081054617600	187081054617600	
Cosine	0	0	
Sine	0	0	
			+
Multiply	1670830136832	1670830136832	
Add	188402145220608	188402145220608	
Cosine	15364608	307292160	
Sine	15364608	307292160	

Time@AMD 1900+: **118796 seconds**
Real-time calibration requires at least **19.0 TFlops**.

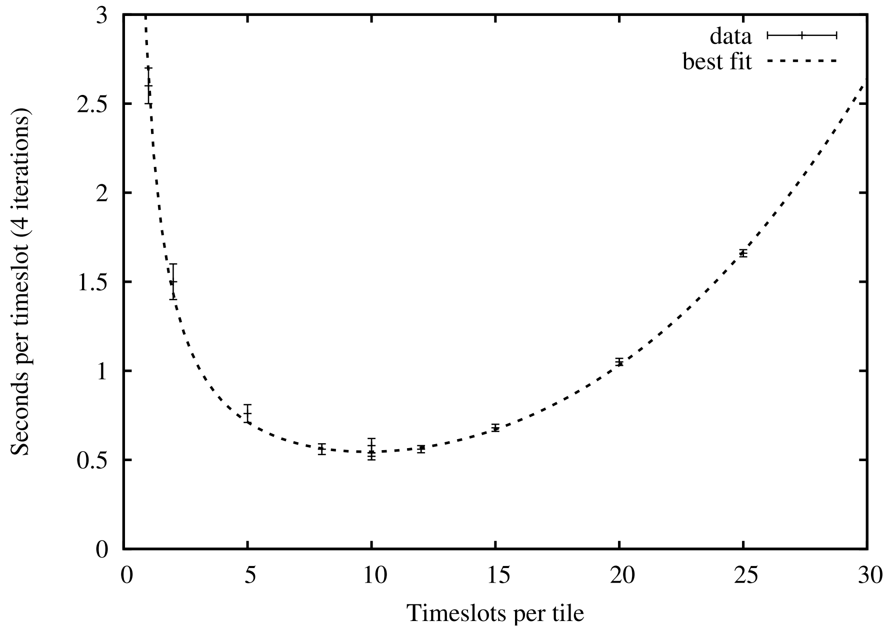

Figure 1: Measured execution time per timeslot of MeqTree system.

9 Comparison with MeqTree system

In order to compare the runtime prediction to the actual execution time of the MeqTree system, I did a timing experiment. The execution time per timeslot solution was measured as a function of the number of timeslots that were solved simultaneously in the same solver instance.

This approach enabled a decomposition in several distinct execution time contributions. First there is a fixed overhead involved with , e.g., function calls. This overhead is constant per solve domain. Its contribution per timeslot is proportional to 1/(number of timeslots). Then there is the inversion of the covariance matrix in the solver. Its cost is proportional to the third power of the number of solvable parameters. As the number of solvables is linearly proportional to the number of timeslots in a solution domain, the contribution of the matrix inverse is proportional to the number of timeslots cubed. Last but not least there is the fixed amount of work that needs to be done for every single timeslot. It consists of

- The actual predict, including any inefficiencies
- Adding normal equations to the solver
- I/O and data handling
- Something else?

In this experiment we can not discriminate between these contributions.

The measurements are plotted in Fig. 1. The dashed line is the best least squares fit of Eqn. (10) to the data.

$$t = (\frac{a}{x} + bx^3 + c)n_{\mathrm{iter}} \tag{10}$$

The best fit coefficients are $a = 0.60 \pm 0.03$, $b = 2.13 \pm 0.05 \times 10^{-5}$, $c = 0.059 \pm 0.004$. For our comparison, coefficient c is the most interesting. Referring back to Sec. 8.1, one sees that at least 0.00545 s are required for the predict on an AMD1900+, which coincidentally is precisely the machine on which this experiment was run.

For four iterations, we see that the constant amount of time spent per timeslot is $0.059 \times 4 = 0.24$ s. This is 43 times slower than the minimum computed in Sec. 8.1. At the moment we do not know what the contribution of adding the equations is.

We *are* able to estimate the amount of time spent in disk I/O. The dataset is 1.3 GB in size. If we assume that the effective disk I/O rate is of the order of 20 MB per second, *and* we consider that the data are also corrected and written back to disk, then a grossly pessimistic estimate of the disk I/O contribution is: 0.09 seconds per timeslot. (read AND write full 1.3 GB, 1440 timeslots in total).

Subtracting this disk-I/O leaves 0.15 seconds for computing. If we furthermore assume that one can attain effectively 50% of the peak performance of the CPU, then we are down to 0.07 seconds. Still almost a factor of 13 more than the estimate in Sec. 8.1.

When analyzing the tree profiling data we consistently see that the solver consumes most of the CPU cycles. We snevertheless still need to investigate whether that is caused by adding the normal equations or by other, constant-per-timeslot overhead in the solver.

Using coefficient b of Eqn. (10), we can estimate the actual cost of the matrix inversion inside the Aips++ Levenberg-Marquardt solver. For n solvable parameters, the matrix inversion costs: $1.14 \times 10^{-9} n^3$ s on an AMD 1900+ CPU. That corresponds to about $1.82 \times n^3$ CPU cycles.

If there are 77 stations and only 2 coefficients are solved for per matrix inversion, then one matrix inversion would cost about 6.6 million CPU cycles. On average, four iterations are needed to converge. Assuming 20 patches, and considering that we need to to this four times per matrix in order to determine the other coefficients, and multiplying the result by 200 channels/subbands, we find that in this case 420 Gcycles are needed to do all the matrix inversions for one timeslot. If one cycle corresponds to one Flop, 420 GFlops should be reserved for matrix inversions in the solver. This does not include adding the normal equations, which could possibly be far more expensive than the relatively small matrix inversions.

The makeNorm equation we use:

```
template <class U, class V, class W>
void LSQFit::makeNorm(uInt nIndex, const W &cEqIndex,
                      const V &cEq, const U &weight,
                      const U &obs,
                      Bool doNorm, Bool doKnown) {
  if (doNorm) {
```

```
    for (uInt i=0; i<nIndex; ++i) {
      if (cEq[i] != 0) {
        Double *i2 = norm_p->row(cEqIndex[i]);  //row pointer
        Double eq(cEq[i]);
        eq *= weight;
        for (uInt i1=0; i1<nIndex; ++i1) {
          if (cEqIndex[i]<=cEqIndex[i1] && cEq[i1] != 0) {
            i2[cEqIndex[i1]] += eq*Double(cEq[i1]); //equations
          };
        };
      };
    };
    state_p &= ~TRIANGLE;
  };
  if (doKnown) {
    Double obswt = obs*weight;
    for (uInt i1=0; i1<nIndex; ++i1) {
      if (cEq[i1] != 0) {
        known_p[cEqIndex[i1]] += Double(cEq[i1])*obswt; //data vector
      };
    };
    error_p[NC] += 1;                           //cnt equations
    error_p[SUMWEIGHT] += weight;               //sum weight
    error_p[SUMLL] += obs*obswt;                //sum rms
  };
}
```

10 A bit more realistic

Evidently, the amount of computing power needed for the self calibration is astonishing. In order to obtain a more realistic number for the required amount of TFlops, one should take into account the cost of the makeNorm() function call, approximately 50% CPU efficiency, and the duty cycle for the given experiment.

An estimate for the cost of one makeNorm call is of the order of 50 cycles for the call itself, and 7 times the number of derivatives for a given datapoint squared. There are typically $4P$ derivatives per data point, so we would have a cost of the order of $112P^2$ cycles per data point. In total this needs to be done BR times per iteration per beam. Therefore the contribution of makeNorm to a full lofar with 20 patches and 200 channels that would amount to 420 GFlops at 100% efficiency.

Therefore an estimate for the selfcal cost of LOFAR could be: (4.7 TFlops + 0.4 TFlops(makeNorm) + 0.4 TFlops(solver matrix invert))×2(efficiency)×2(anything we forgot business) = 11 TFlops.

For the EOR a similar estimate would be: 19 +

11 How to reduce cost

Polynomials help in reducing the cost of evaluating the measurement equation. In fact, they make matters worse. Instead of "one" derivative per datapoint, one must compute "N_{coeff} derivatives, effectively multiplying the required processing power by the degree of the polynomial plus one.

A coarser data grid is the easiest structural thing that helps considerably. It is already evident that it would be rediculous to calibrate on a 1 second, 10 kHz grid. Instead I propose to calibrate per subband in 1 s time intervals for the full LOFAR. PErhaps we even need to go to 10 s timeslots. In any case we must properly account for time- and bandwidth smearing in the presence of missing data points at a finer grid than the calibration grid.

Correction should nevertheless be performed on a finer grid. Here it might help to use some form of polynomial interpolation for accuracy. The correction step is probably much cheaper because it needs only be done once per beam per timeslot and not for all derivatives and iterations.

One can also reduce the number of baselines in the solution:

- no short baselines: get rid of Milkyway / large scale, hard to model stuff
- no long baselines: Little signal anyway because sources are resolved.

12 Conclusions

In terms of sheer FLOPS, calibrating the full LOFAR in real time is going to be extremely challenging. Chances are much better for the EOR experiment that only uses 32 stations and 10 seconds integration time in combination with a low dutycycle. All in all I think that ways can be found to do the calibration in almost real time for reasonable processing cost. We may however need to make the calibration cluster a bit more powerful than initially intended.