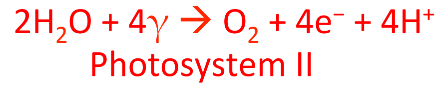
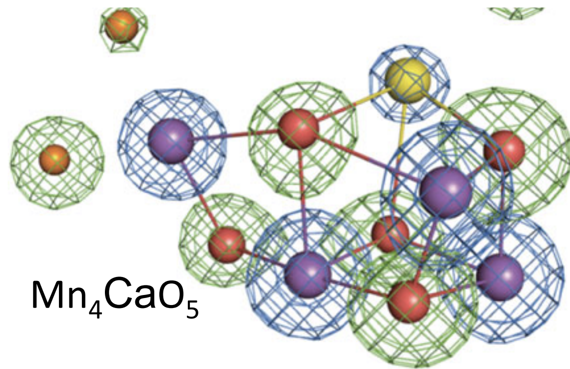
6th DIALS Workshop: Deployment at Light Source Facilities
LBNL, Wednesday, May 27, 2015

# *Large Parameter Optimizations*


Nicholas K. Sauter
Lawrence Berkeley National Laboratory

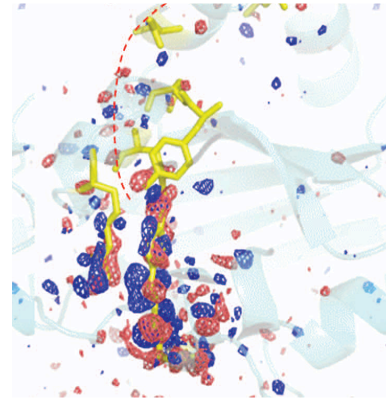
# X-ray free-electron laser rationale



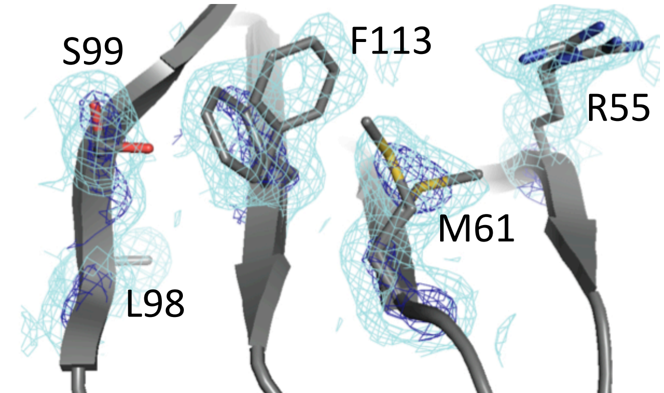

Avoid radiation damage

Photoactive yellow protein
Tenboer et al (2014) Science

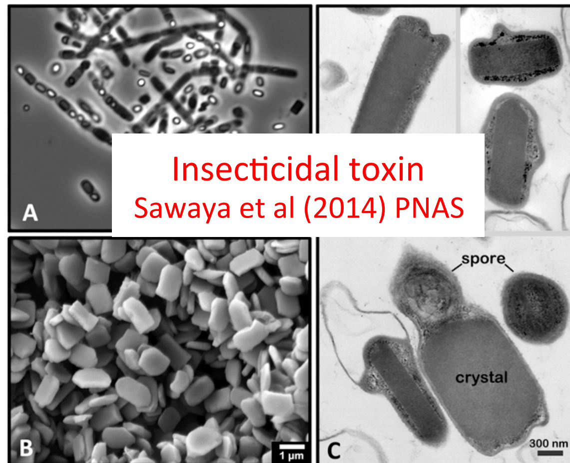


Time domain

Cyclophilin A conformational switch
Keedy et al (2015) submitted

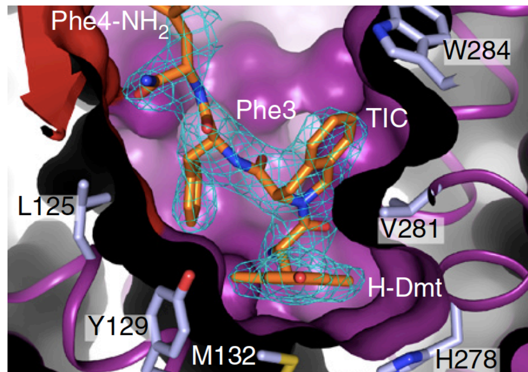



Room temperature science

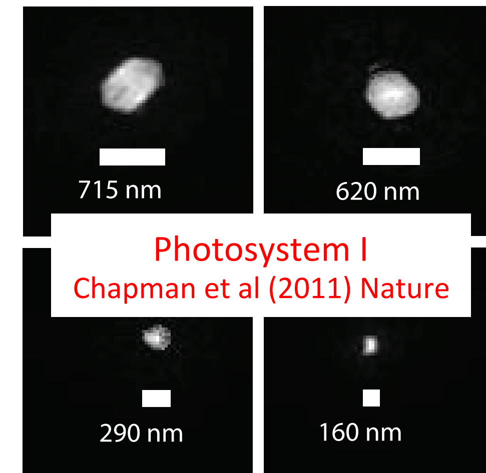



In vivo science

Opioid receptor + antagonist
Fenalti et al (2015) Nat. Struct. Mol. Biol.

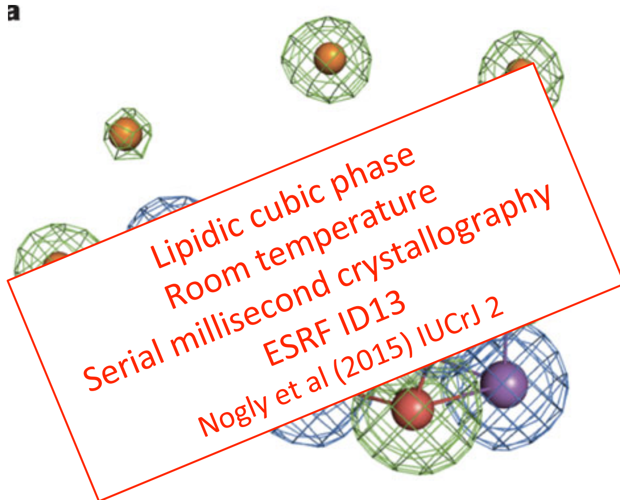



Extended resolution




Nanocrystals

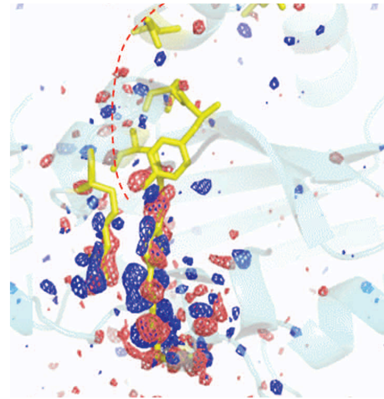
# Still shots / serial crystallography rationale



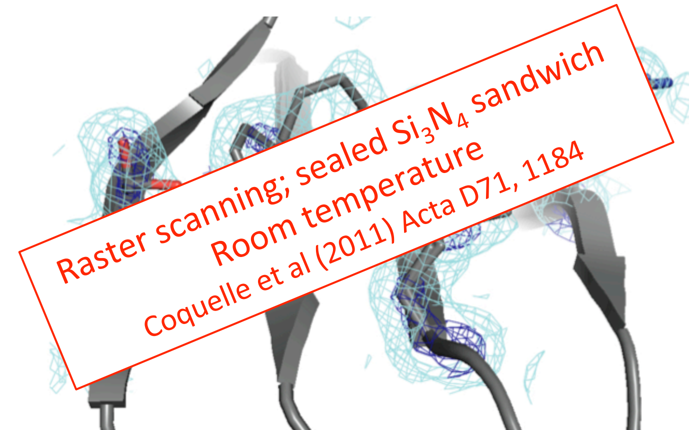

Avoid radiation damage

Photoactive yellow protein

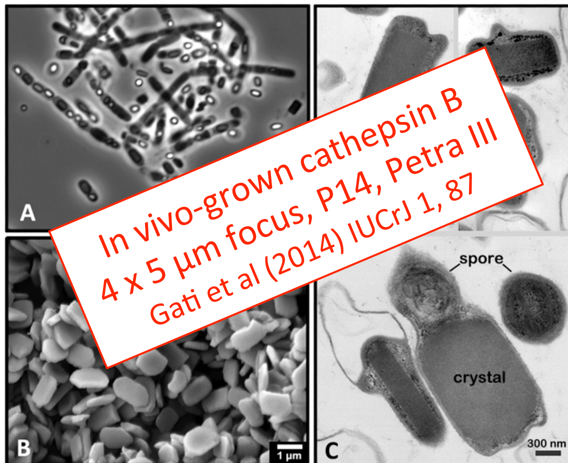
Tenboer et al (2014) Science



Time domain



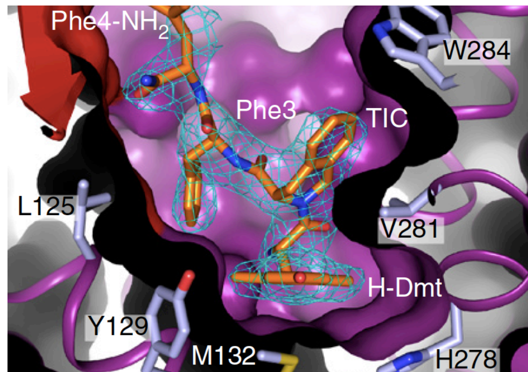

Room temperature science



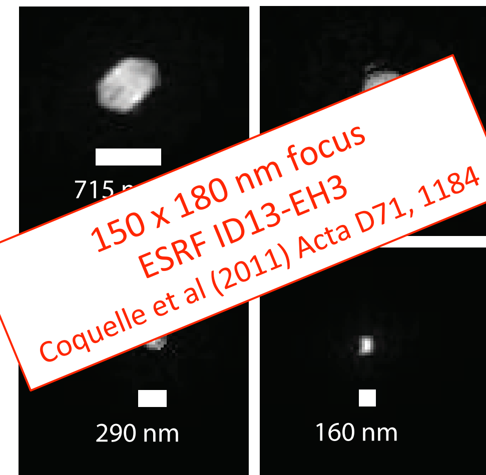

In vivo science

Opioid receptor + antagonist

Fenalti et al (2015) Nat. Struct. Mol. Biol.

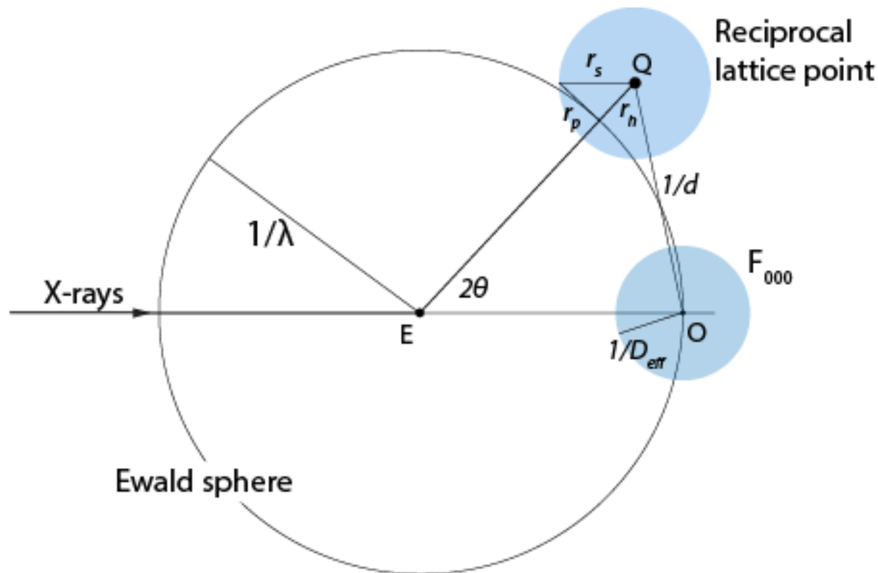



Extended resolution




Nanocrystals

# Still shots: mathematical expression for spot partiality is a work in progress




$$\text{Partiality}_{\lambda} = \frac{\text{area / volume}_{HKL}}{\text{area / volume}_{000}}$$

$$\Psi = \sum_{\textbf{observations}} (I_{\textbf{observed}} - I_{\textbf{model}})^2$$

- This correction is extremely sensitive to orientation
- Crystal must be postrefined: the orientation is optimized such that the partiality-corrected multiple measurements of the same HKL yield the most consistent structure factor.
- Papers by Helen Ginn & Mona Uervirojnangkoorn
- Sauter (2015) *J. Synchro. Rad.*, 22, 329

# Scale factor refinement is analytically complex

$$F = \sum_h \sum_i W_{hi}(I_{hi} - G_m I_h)^2$$

G = per-image scale factor
I = s.f. intensity

$$I_h = \left( \sum_i W_{hi} G_m I_{hi} / \sum_i W_{hi} G_m^2 \right).$$

$$F = \sum_h \sum_i W_{hi}(I_{hi} - G_m \frac{\sum_i W_{hi} G_m I_{hi}}{\sum_i W_{hi} G_m^2})^2$$

$$\frac{\partial F}{\partial G_m} = 2 \sum_h \sum_i W_{hi}(I_{hi} - G_m \frac{\sum_i W_{hi} G_m I_{hi}}{\sum_i W_{hi} G_m^2}) * (- \frac{\partial G_m \frac{\sum_i W_{hi} G_m I_{hi}}{\sum_i W_{hi} G_m^2}}{\partial G_m})$$

- Express $I_h$ in terms of scale factors $G_m$: Hamilton, Rollet & Sparks, 1965
- Derivatives require a double loop over two sets of indices *m* and *m'*
- Introducing partiality requires more parameters, and further analytical complexity

# Alternate approach: Treat the intensities as independent parameters

If the $I_h$ are treated as independent variables the the complexity of the algorithm needed to evaluate the derivatives is linear in the number of observations n *O(n)*.

$$F = \sum_i W_{hi}(I_{hi} - G_m I_h)^2 \text{........}(4)$$

$$\frac{\partial F}{\partial G_m} = 2\sum_i t(-I_h W_{hi}) \text{............}(5)$$

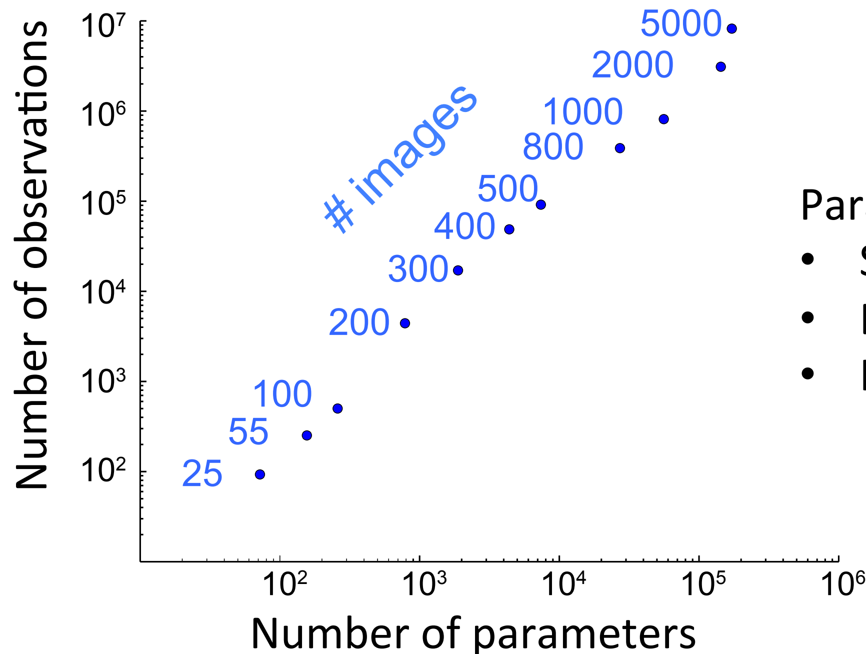
$$\frac{\partial F}{\partial I_h} = 2\sum_i t(-G_m W_{hi}) \text{............}(6)$$

where $t = I_{hi} - G_m I_h$

# Model the problem: scale thousands of synthetic-data still shots

- Take Miller indices from actual indexed images with real diffraction
- Substitute calculated intensities from a PDB model, with various degrees of shot noise
- Use full intensities only – do not simulate the partiality problem
- Choose randomized scale factors and B-factors for each image

$$T = \sum_{\text{obs}} \frac{1}{\sigma^2} \left( I_{\text{obs}} - G_{\text{image}} e^{-2B_{\text{image}}\left(\frac{\sin\theta}{\lambda}\right)^2} I_{\text{model}} \right)^2$$




Parameters refined:

- Scale factor $G_{\text{image}}$ for each image
- B-factor $B_{\text{image}}$ for each image
- Intensity $I_{\text{model}}$ for each Miller index

# Key methodology for refinement

- Use a limited-memory quasi-Newton method (LBFGS) to minimize the target function $T$.
- For each step, calculate the objective function and the first derivatives:

$$T$$

$$\frac{\partial T}{\partial G_i}, \ldots, \frac{\partial T}{\partial B_i}, \ldots, \frac{\partial T}{\partial I_i}, \ldots$$

- Straightforward calculation of starting values:
  - Scale factor G: average intensity of all observations on each frame
  - B factor = 0
  - Intensity I: average intensity of all observations of the Miller index

# Key methodology for refinement

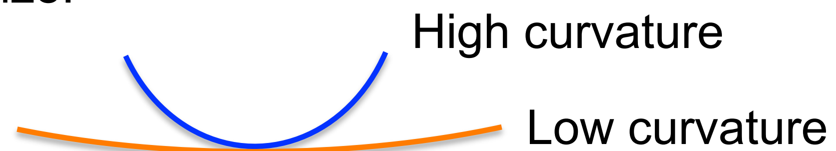
- Use a limited-memory quasi-Newton method (LBFGS) to minimize the target function $T$.
- For each step, calculate the objective function and the first derivatives:

$T$

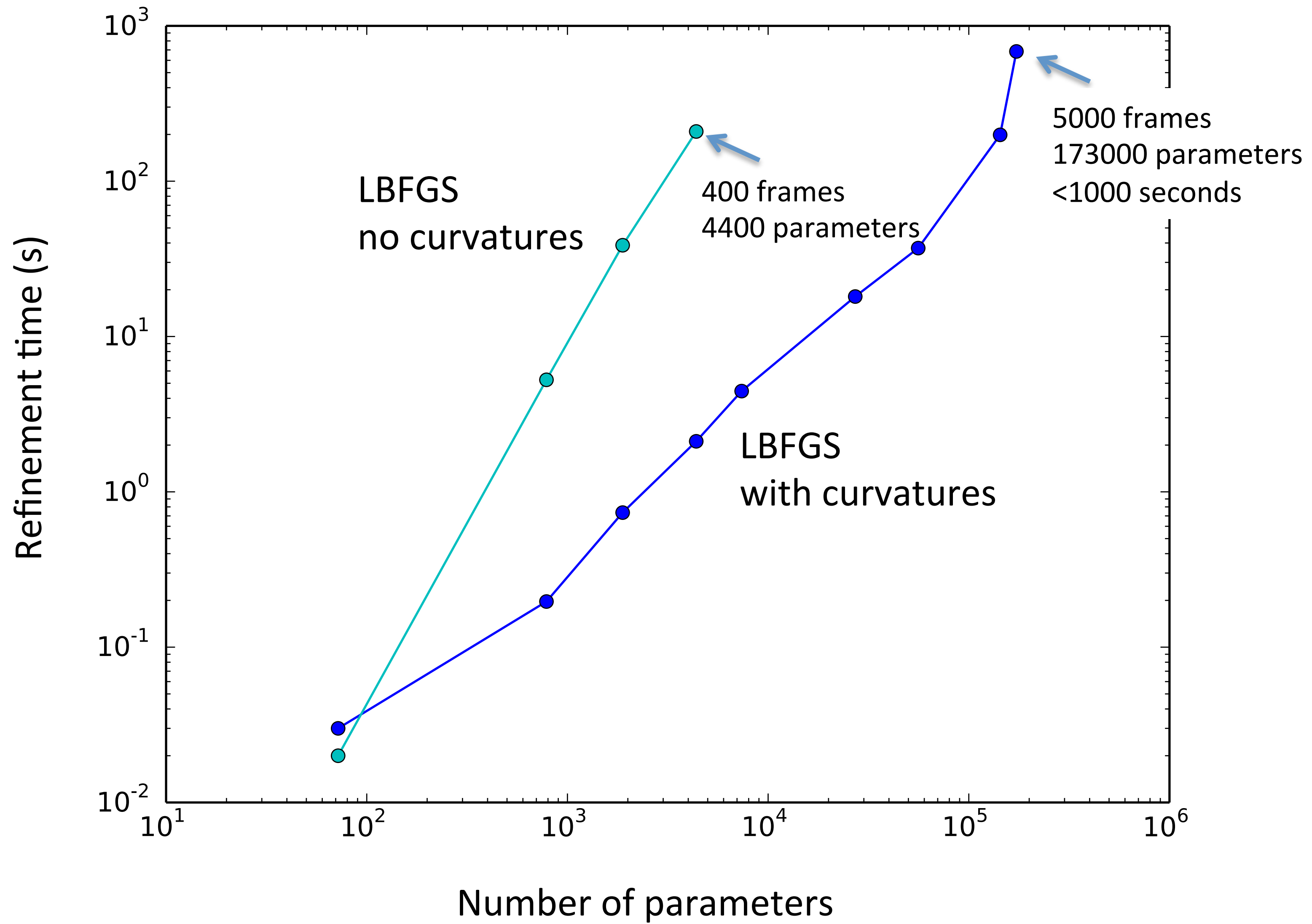
$$\frac{\partial T}{\partial G_i}, \ldots, \frac{\partial T}{\partial B_i}, \ldots, \frac{\partial T}{\partial I_i}, \ldots$$

- However, the convergence behavior is extremely poor, presumably due to the large number of parameters.
- *e.g.* 400 frames with 4000 parameters requires $10^5$ iterations and 417 seconds.
- Fix this by computing curvatures. Curvatures condition the problem, properly weighting the contribution of each free parameter to the target function, and arriving at a more appropriate step size:

$$\frac{\partial^2 T}{\partial {G_i}^2}, \ldots, \frac{\partial^2 T}{\partial {B_i}^2}, \ldots, \frac{\partial^2 T}{\partial {I_i}^2}, \ldots$$



# Performance of LBFGS + / - curvatures

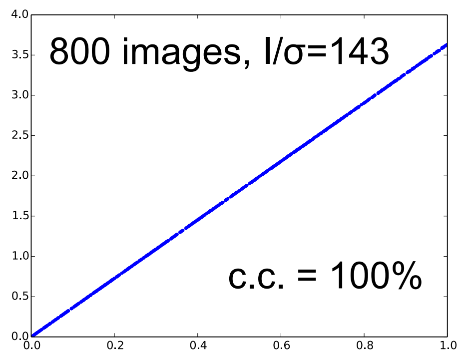


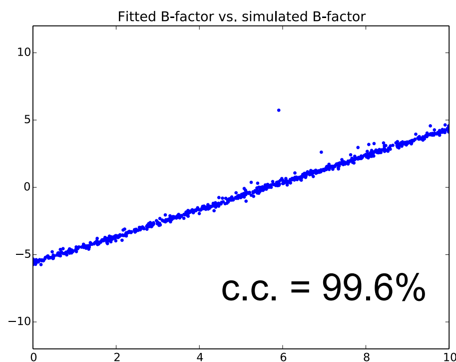
# Refinement results

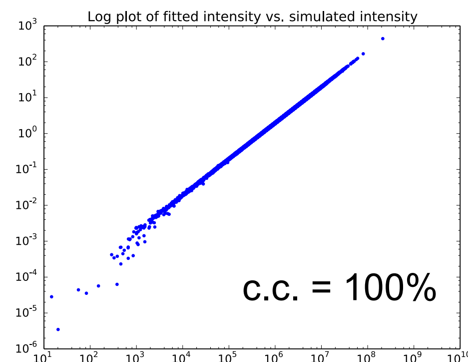
## Scale factors

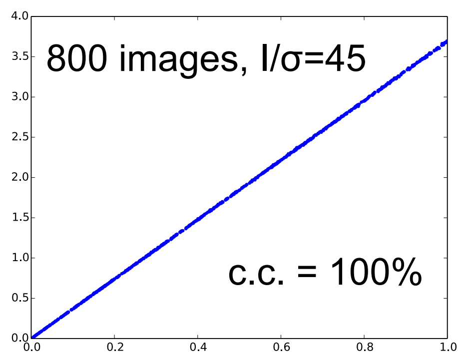
## B factors

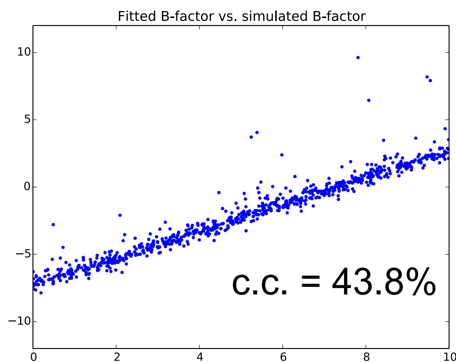
## Intensities

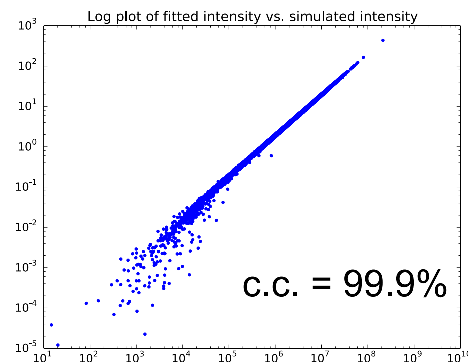


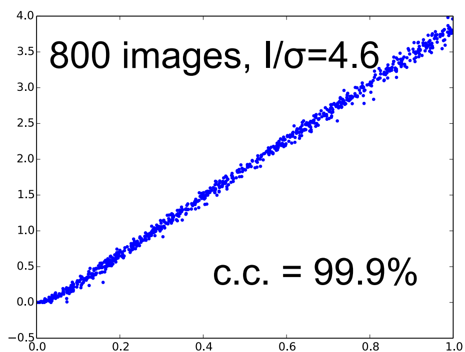



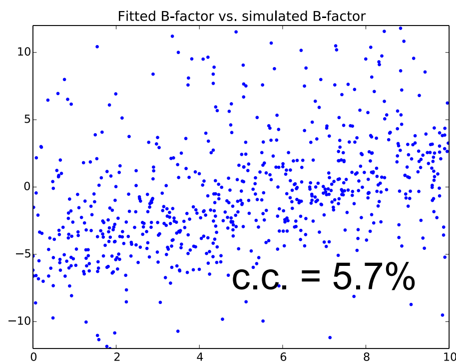



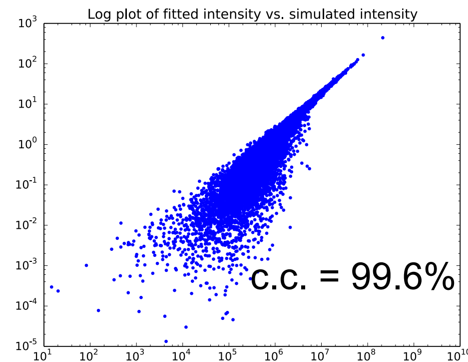



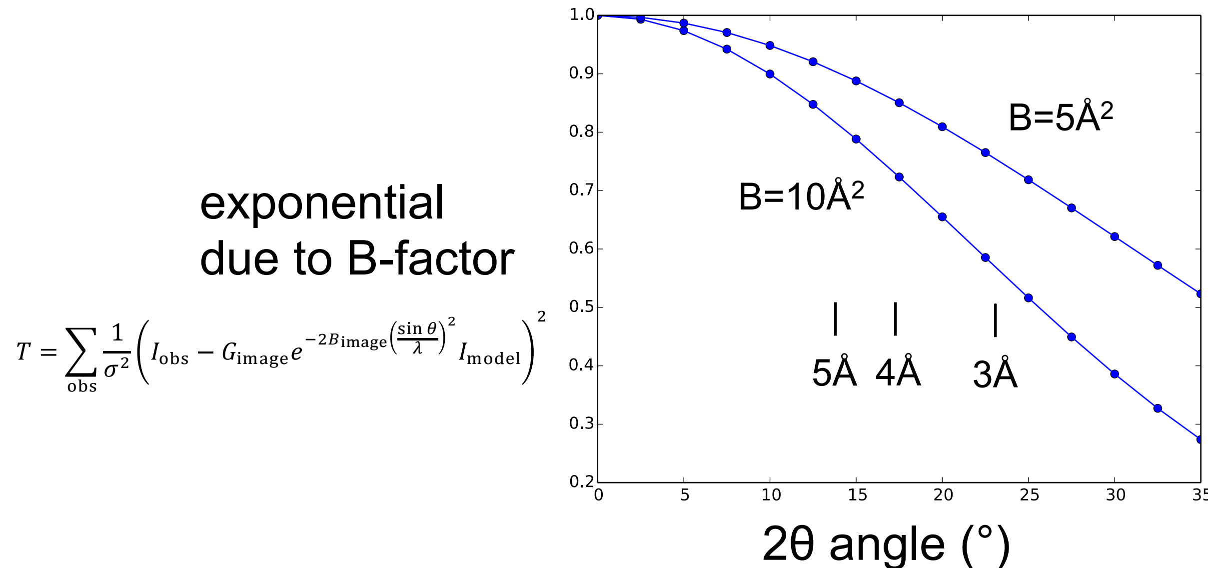



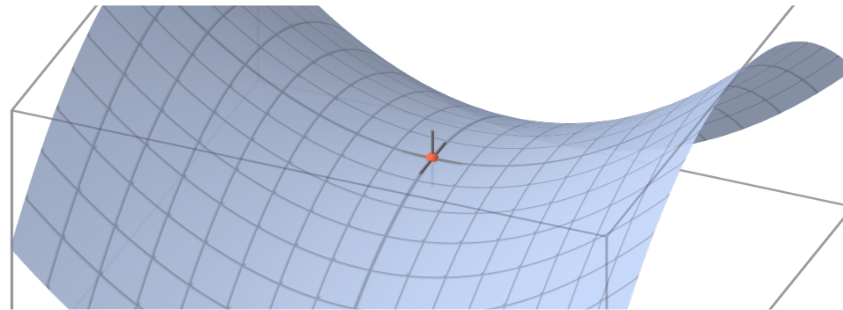
# B-factor results are inherently noisy




- Exponential factor makes relatively little contribution to the target function
- Same comment applies to the weak structure factor intensities; potentially applies to other model parameters going in to postrefinement
- Confirmed by the follow-up experiments with a different optimizer, but using the same target function. Minimization did converge, fit was just poor.

# LBFGS with curvatures is not a good general method

- Analytical calculation of second derivatives is labor-intensive and error-prone
- Makes the code more complex, and more difficult to maintain
- Discourages quick experiments with different parameters or alternate models, which is precisely the aim of this inquiry
- Our simple guess of B=0 as the initial parameter value puts us at a saddle point for some of the B-factors (pointed out by Muhamed/DIALS5)



- Unless the curvatures are positive for all parameters, the LBFGS minimizer does not know how to proceed.

# Workarounds for non-positive curvature are not robust

- Muhamed applied pre-refinement of I, G, and B without curvatures, to put the problem at a better position in parameter space where curvatures could then take over.

- This turned out to still fail for a small set of B-factors in my simulations, further illustrating that we want to avoid complex, *ad hoc* methods, and rather focus on simple methods that can be generalized to other parameter sets in the future.

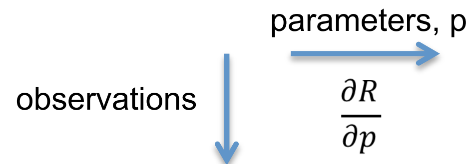
- Results presented here: avoided negative curvatures by taking the absolute value!
  - Allowed the minimization to run to completion
  - However, invariably took false directions where the exponential term diverges to infinity
  - Only able to run the program by turning off the trap for floating point exceptions
  - NOT a general method
  - B-factors for low I/sigma examples do not always converge

# Turn to non-linear least squares methods

- Optimally take advantage of the least-squares form of the target.

$$T = \sum_{\text{obs}} \frac{1}{\sigma^2} \left( I_{\text{obs}} - G_{\text{image}} e^{-2B_{\text{image}} \left( \frac{\sin\theta}{\lambda} \right)^2} I_{\text{model}} \right)^2$$

- Jacobian matrix: for each observation, calculate the derivative of the residual term $R_{obs} = I_{obs} - I_{calc}$ with respect to each free parameter.




- These terms are used to build up the so-called normal equations $Ax = b$, whose solution gives the next step for iterative non-linear least squares parameter fitting. $A = J^T J$

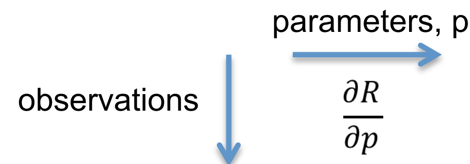
- Levenberg-Marquardt algorithm arrives at a good fit in very few iterations, with information from first derivatives only.

# Problems to overcome

- Optimally take advantage of the least-squares form of the target.

$$T = \sum_{\text{obs}} \frac{1}{\sigma^2} \left( I_{\text{obs}} - G_{\text{image}} e^{-2B_{\text{image}} \left( \frac{\sin\theta}{\lambda} \right)^2} I_{\text{model}} \right)^2$$

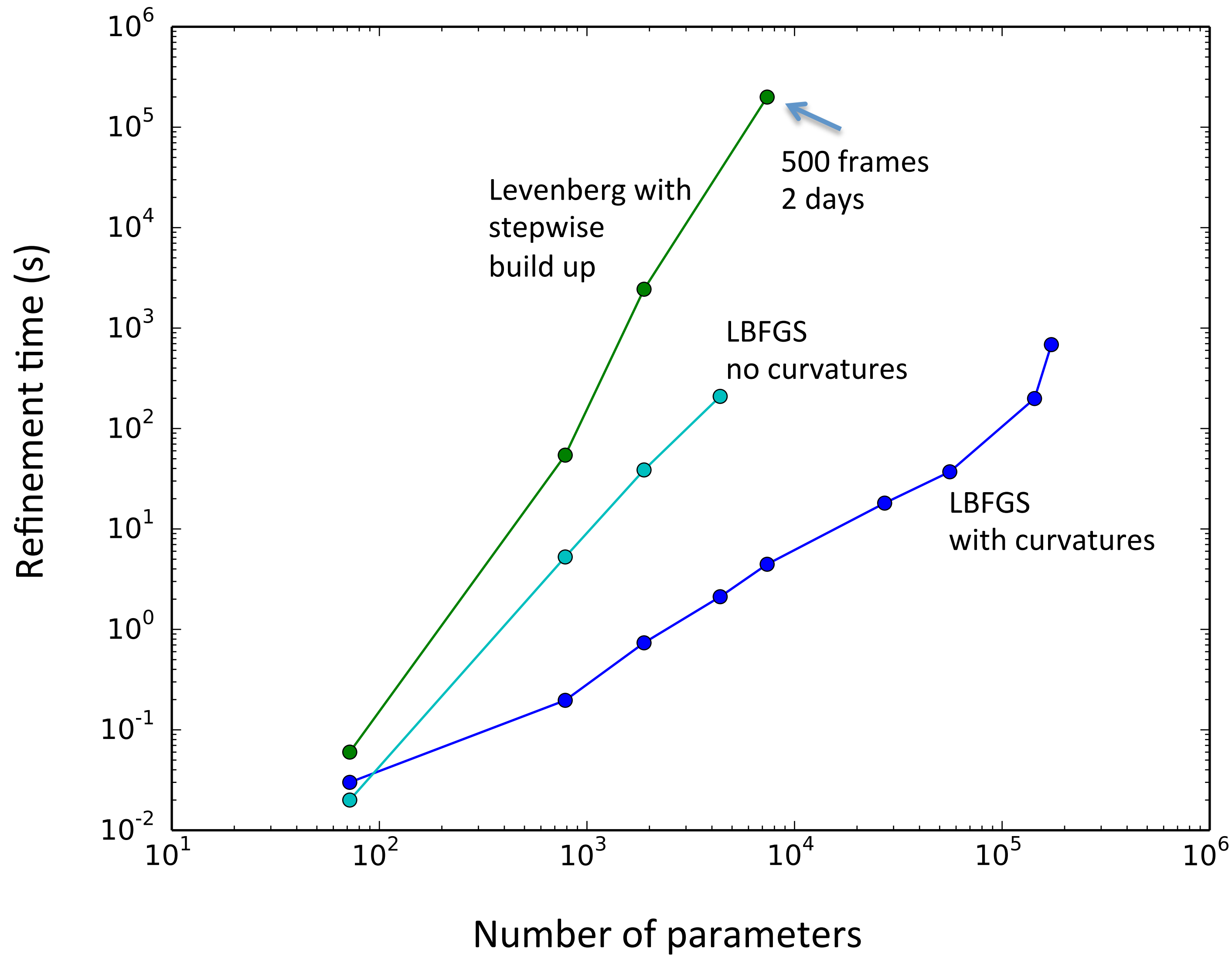
- Jacobian matrix: for each observation, calculate the derivative of the residual term $R_{obs} = I_{obs} - I_{calc}$ with respect to each free parameter.




- The Jacobian matrix is notoriously large. For the 5000-image problem considered here, 8 x $10^6$ observations = 1.7 x $10^5$ parameters, or $10^{12}$ terms. This could not possibly fit in memory.

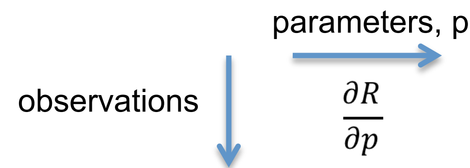
- However, each line of the Jacobian can be calculated sequentially and added to the normal equations. Furthermore, perform this loop at the C++ level, avoiding inefficient Python code.

# Performance of non-linear least squares



# Further improvements to efficiency

- Jacobian matrix: for each observation, calculate the derivative of the residual term $R_{obs} = I_{obs} - I_{calc}$ with respect to each free parameter.



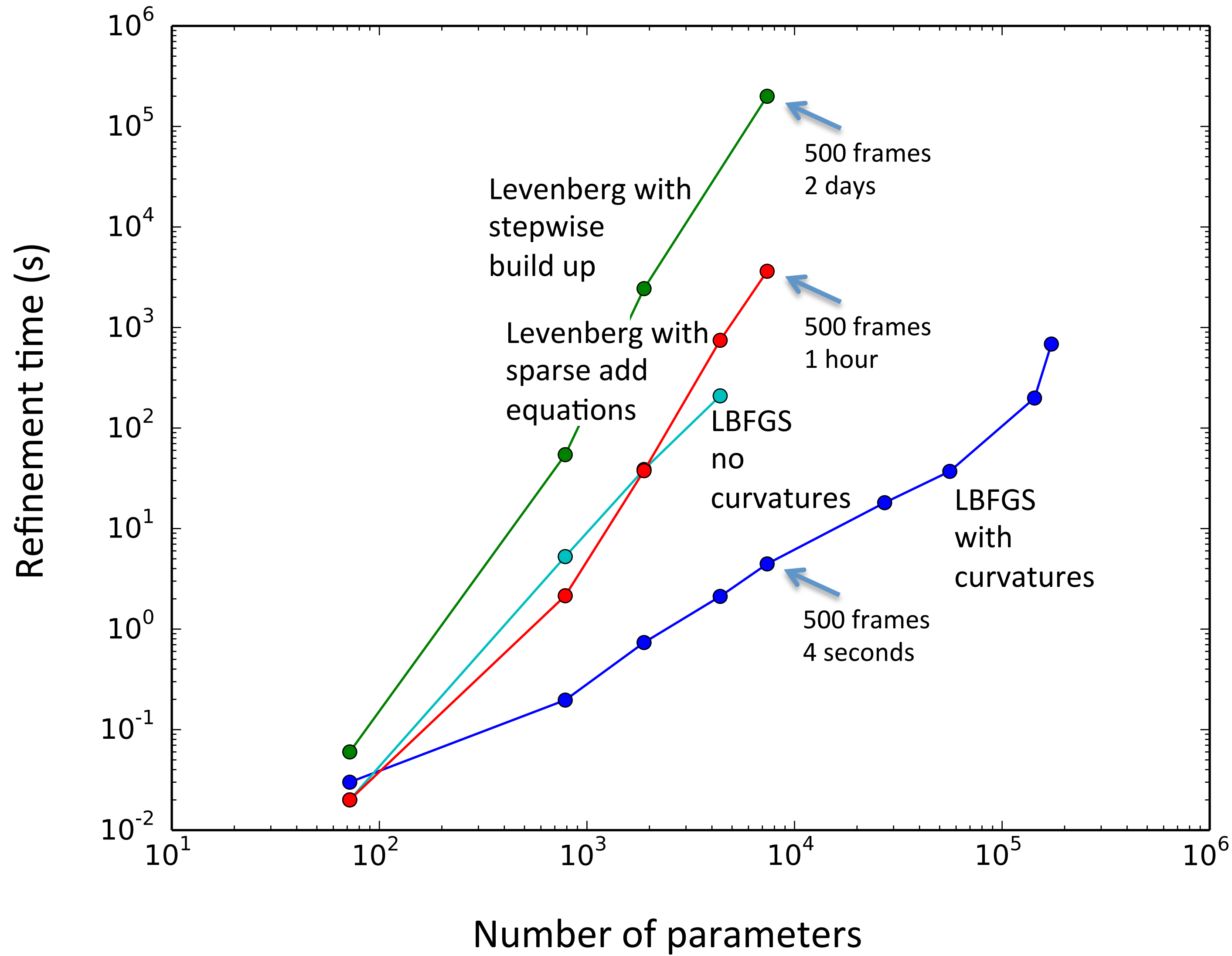

$$\frac{\partial R}{\partial p}$$

- For each row in the Jacobian matrix, only three elements are non-zero: the derivatives of the residual with respect to $G_{image}$, $B_{image}$, and $I_{miller}$. Therefore, loop over these three explicitly when building up the normal equations, instead of over thousands of zeroes.

```
for (int ix = 0; ix < raw_observations.size(); ++ix) {
        double Gfactor  = Gptr[ frame[ix] ];
        double Bfactor  = std::exp(-2.* Bptr[ frame[ix] ] * stol_sq[ix]);
        double Ifactor  = Iptr[ miller[ix] ];

        jacobian_one_row_indices.push_back( miller[ix] );
        jacobian_one_row_data.push_back(-Bitem * Gitem); //derivative with respect to I

        jacobian_one_row_indices.push_back(N_I + frame[ix]);
        jacobian_one_row_data.push_back(-Bitem * Iitem); //derivative with respect to G

        jacobian_one_row_indices.push_back(N_I + N_G + frame[ix]);
        jacobian_one_row_data.push_back(Bitem * Gitem * Iitem * 2. * stol_sq[ix]); //derivative with respect to B

        add_residual(residuals[ix], weights[ix]);
        add_equation_sparse(-residuals[ix], jacobian_one_row_indices, jacobian_one_row_data, weights[ix]);
}
```

- Specific code for this parameterization, down to the add_equation level, but after all this is only 13 lines of code, so it's worth it!

# Performance of non-linear least squares



# How can we break the 500 frame/1 hour limit?

- Idea 1: divide and conquer.

  - cspad.metrology. Macrocycle iteration over two refinements:
    - Tile positions & rotations, beam position, crystal distance, and crystal $Rot_z$ (done by LBFGS + curvatures)
    - Individual Levenberg refinement for each crystal of unit cell, and crystal rotations $Rot_x$ and $Rot_y$.

  - cctbx.prime. Microcycle iteration over:
    - G (scale) factors
    - Crystal orientation
    - Mosaic parameters
    - Unit cell dimensions

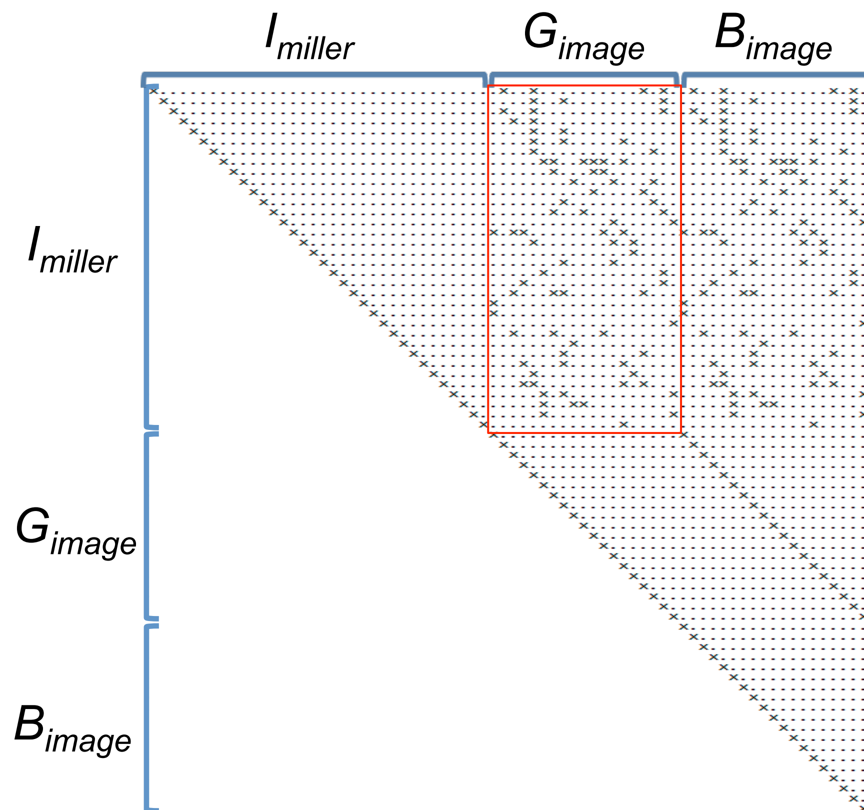
    Macrocycle over:
    - Reference intensities

- The drawback to these approaches is that independent refinement of parameters ignores covariance.

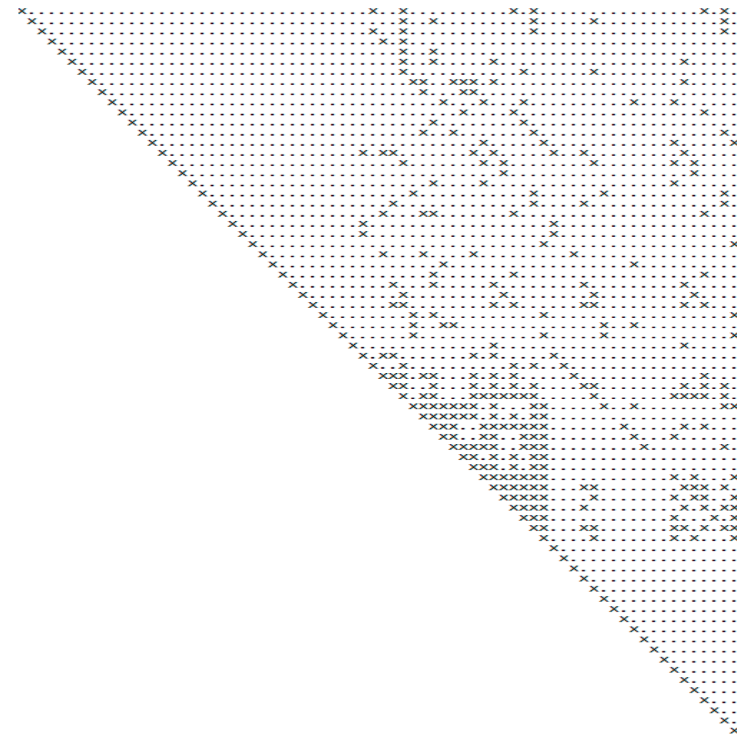
- To facilitate research into new parameterizations, how far can refinement be pushed without dividing parameters into separately-refined piles?

# Examining the normal matrix

- Idea 2: Use better algorithms to solve large normal matrices *A*.
  - Solving $Ax = b$ gives the step size for Levenberg-Marquardt iterations
  - *A* is a positive-definite symmetric square matrix, $N_{params} \times N_{params}$
  - Rate limiting step in calculating $x = A^{-1}b$ is the use of the Cholesky decomposition of *A* into the product $A = LL^T$, where *L* is a lower-triangular matrix, or “Cholesky factor”.

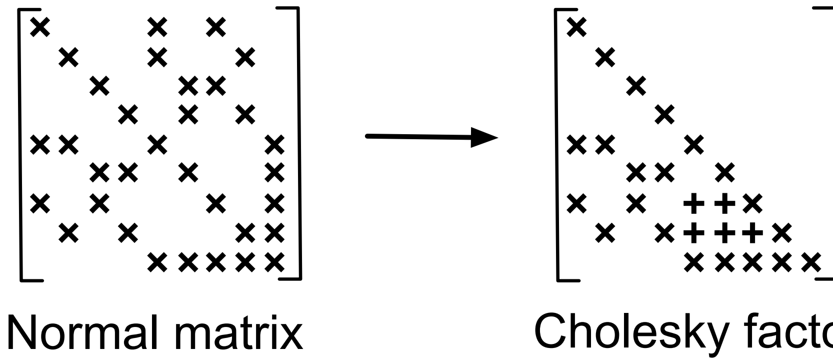


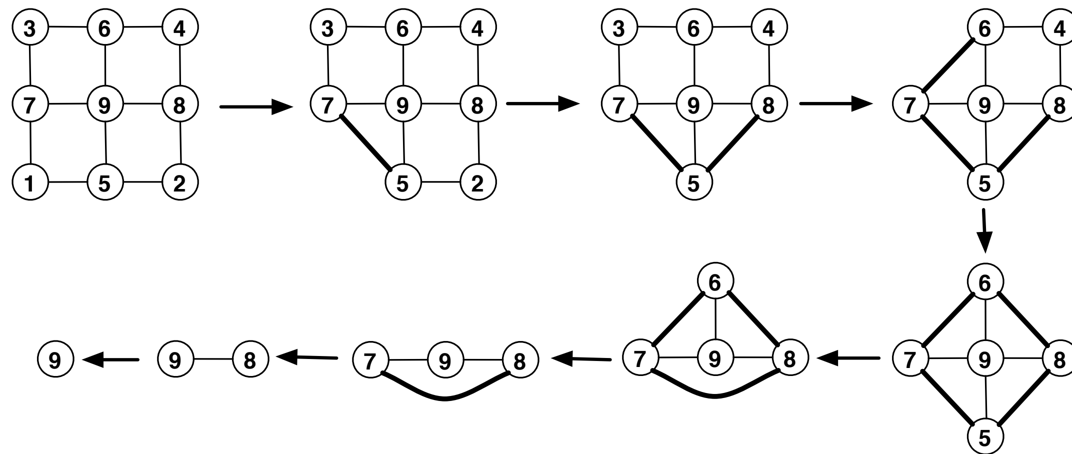

Example normal matrix



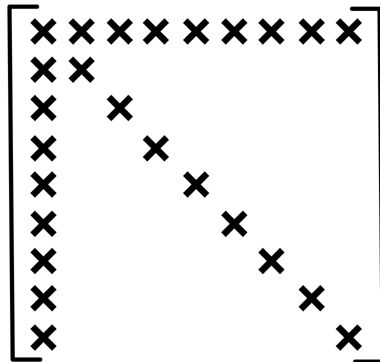
Corresponding Cholesky factor

# Analytically compute the Cholesky fill-in with graph theory

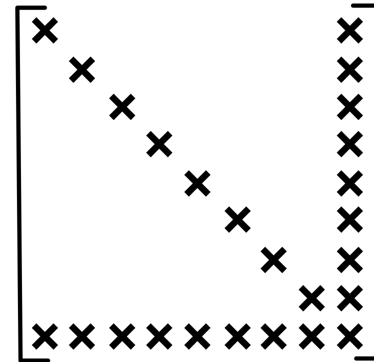


# Fill-in can be minimized by permuting the order of parameters



100% fills



zero fills

# Significant efficiency benefit: only calculate the non-zero elements

| # images | # parameters | $N^2$ | # non-zero elements in normal matrix | Fraction of elements non-zero | # non-zero elements in Cholesky factor | Fraction of elements non-zero |
|---:|---:|---:|---:|---:|---:|---:|
| 200 | 786 | $6x10^5$ | 9,565 | 2.97% | 95,247 | 29.58% |
| 300 | 1,884 | $4x10^6$ | 35,611 | 1.97% | 213,967 | 11.82% |
| 500 | 4,383 | $2x10^7$ | 188,428 | 0.69% | 681,931 | 2.49% |
| 1000 | 55,964 | $3x10^9$ | 1,672,461 | 0.11% | 3,618,497 | 0.23% |
| 2000 | 143,279 | $2x10^{10}$ | 6,310,536 | 0.06% | 14,171,257 | 0.14% |
| 5000 | 172,695 | $3x10^{10}$ | 16,509,042 | 0.11% | 66,346,347 | 0.44% |

# Optimization package pros & cons

Pros:

- Very large community effort, popular
- Many algorithms supported

Cons:

- Scipy.optimize.leastsq depends on fortran
  - Memory contention forces special python import order
- Requires the source developer to install fortran compiler
- Very difficult for us to prepare portable multiplatform binary distribution
- Cython compiler is required
- Lots of other dependencies—ATLAS, LAPACK etc.
- Levenberg-Marquardt already in scitbx
- Only supports sparse-matrix Cholesky with additional package scikits.sparse.cholmod (GPL?)

Eigen

Pros:

- Very large community effort, actively developed
- Many linalg algorithms supported
- Based solely on C++ template library
  - No additional compilers needed
  - No compiled libraries; operates exclusively through include files
- Permissive license

Cons:

- No Python interface
  - But the relevant algorithms are easily exposed (40 lines of code) through our normal boost-wrappers; 2-day effort, no attempt to provide to-python/from-python wrappers for Eigen data structures; LBNL to provide example

# Levenberg-Marquardt numerical results

- Virtually identical to LBFGS results
- Small improvements, especially in B-factor correlation coefficients, where LBFGS curvatures had produced numerical instability

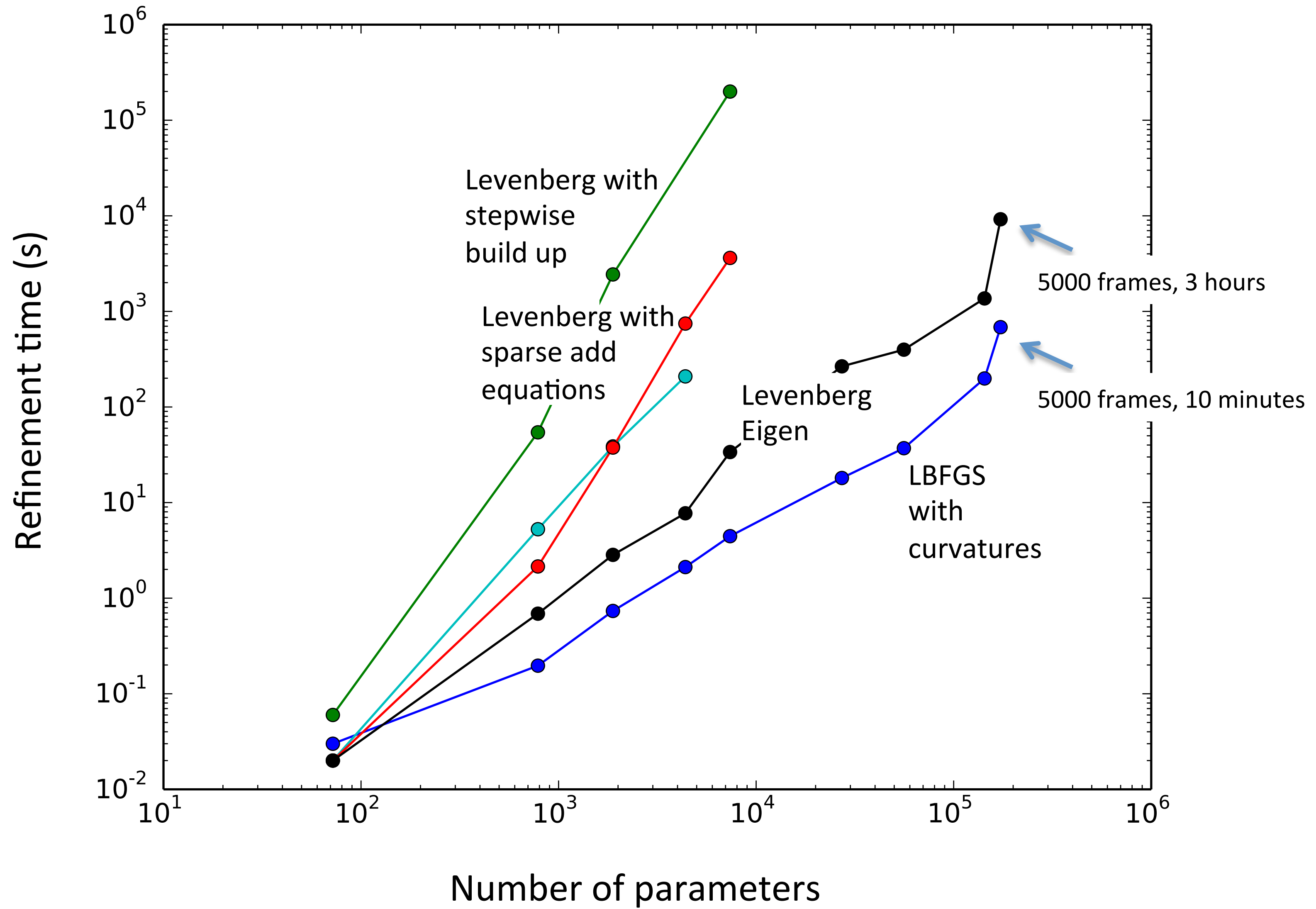
Example:

1000 images, I/sigma=4

LBFGS: G (r=99.8%) B (r=-3.1%) I (99.3%)

L-M: G (r=99.9%) B (r=16.1%) I (99.2%)

# Performance of non-linear least squares



# Next steps

Optimize code:

- Check for memory leaks

Connecting it up to DIALS & XFEL:

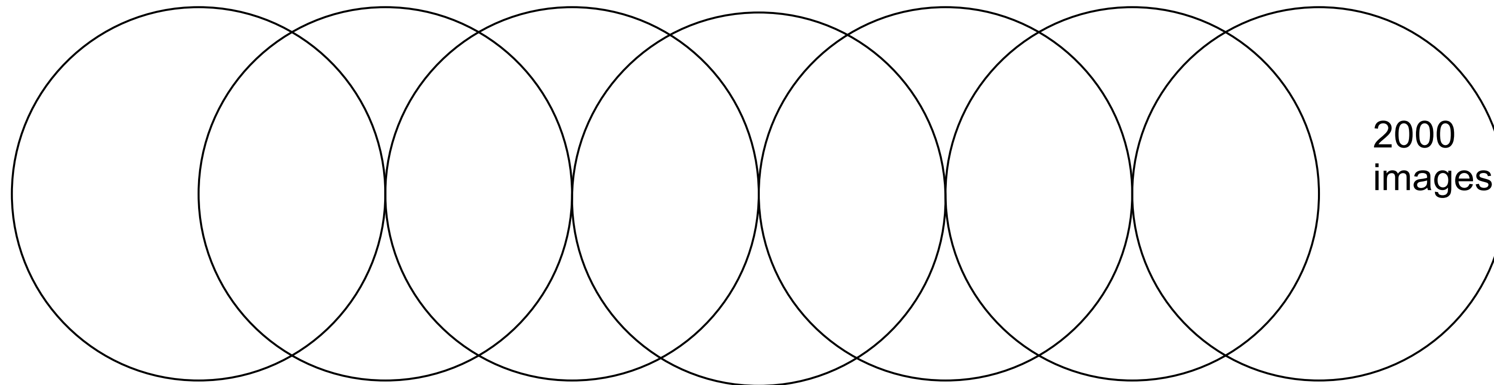
- Include eigen source in the developer build, add include path
- Extend the scitbx/lstbx Python layer to include a sparse Cholesky option
- "How-to" example for writing fast-running build_up() of normal equations (C++)

Applications:

- Drop-in optimizer replacement for cspad.metrology (in the cctbx.xfel context); refine all parameters in one go.
    - Migrate from the ucbp3 spot prediction code to the dxtbx framework
    - Will be a first step toward doing all metrology refinement in DIALS
    - Publish soon
- Drop-in optimizer replacement for the postrefinement option in cxi.merge
    - Experimental platform for adding new partiality models

# Scaling it up

Use “divide and conquer” to go beyond 5000 images




Refine parameters on overlapping sets of 2000 images

Average the duplicate parameter fits & do another round

….on 64 cores, this processes 64,000 images in 1 – 2 hours

Scales up to many nodes if MPI interface is used, to handle $10^5$ images

# DIALS: Diffraction Integration for Advanced Light Sources

Large-parameter outlook:

- Rotation data: application to multicrystal & scan varying datasets
- Enable still-shot methods for synchrotron experiments
- Experimental calibration (metrology)