# Track matching strategies (a fitting talk)

Hasret Nur, Renato Quagliani, **Manuel Schiller**

University of Glasgow, CERN, University of Glasgow

March 7th, 2024

# introduction

- task: match T (SciFi/MP) track segment to Velo to produce long tracks
  - not so difficult if you know track momentum well, and have time to propagate through $\vec{B}$ field
  - idea is to see if we can build a good enough approximation to track propagation that's fast
  - propose a framework to fit arbitrary multi-dimensional approximations
    - not limited to propagating track states through the magnetic field
    - could be used to derive fast momentum parametrisations
    - your application goes here…
- Renato will report on performance of the matching itself etc.
- menu for this talk:
  - fits with model linear in fit parameters (your new secret superpower!)
  - solving resulting equations
  - (multidimensional) Chebyshev expansions
  - fitting approximate track propagators

# fits linear in parameters

- consider $\chi^2 = \sum_k \left(\frac{y_k - m(\vec{x}_k;\vec{p})}{\sigma_k}\right)^2$ where
  - $y$ is what you measure (track position/slope after propagation)
  - $\sigma$ is the uncertainty on your measurement $y$
  - $\vec{x}$ is where you measure (track state from which you propagate)
  - $m(\vec{x};\vec{p})$ is the *model*, with fit parameters $\vec{p}$
  - index $k$ runs over the measurements
- further consider a model that is *linear* in fit parameters: $m(\vec{x};\vec{p}) = \sum_l p_l g_l(\vec{x}) = \vec{g}^T\vec{p}$
  - $g_l$ are arbitrary functions of $\vec{x}$ that do not depend on $\vec{p}$
- special class of models: can solve analytically (on next slide)
  - no need for things like Minuit or RooFit
  - fast to compute (if you choose reasonable $g_l$)

# solving fits linear in parameters

- consider $\chi^2 = \sum_k \left(\frac{y_k - m(\vec{x}_k;\vec{p})}{\sigma_k}\right)^2$
- we put $0 = \nabla_{\vec{p}}\chi^2 = \sum_k \frac{2}{\sigma_k^2}\left(y_k - m(\vec{x}_k;\vec{p})\right)\left(-\nabla_{\vec{p}} m(\vec{x}_k;\vec{p})\right)$
- rewrite: $\sum_k \frac{1}{\sigma_k^2}\vec{g}(\vec{x}_k)\vec{g}^T(\vec{x}_k)\vec{p} = \sum_k \frac{1}{\sigma_k^2} y_k \vec{g}(\vec{x}_k)$

  (if you don't see it, do $\frac{\partial\chi^2}{\partial p_l} = 0$ for some $l$ by hand with pencil and paper)
- abbreviate: $\langle q\rangle = \sum_k \frac{q_k}{\sigma_k^2}$ for some per-measurement quantity $q$

$$\begin{pmatrix} \langle g_0(\vec{x})g_0(\vec{x})\rangle & \langle g_0(\vec{x})g_1(\vec{x})\rangle & \cdots & \langle g_0(\vec{x})g_n(\vec{x})\rangle \\ \langle g_1(\vec{x})g_0(\vec{x})\rangle & \langle g_1(\vec{x})g_1(\vec{x})\rangle & \cdots & \langle g_1(\vec{x})g_n(\vec{x})\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle g_n(\vec{x})g_0(\vec{x})\rangle & \langle g_n(\vec{x})g_1(\vec{x})\rangle & \cdots & \langle g_n(\vec{x})g_n(\vec{x})\rangle \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ \ldots \\ p_n \end{pmatrix} = \begin{pmatrix} \langle y g_0(\vec{x})\rangle \\ \langle y g_1(\vec{x})\rangle \\ \ldots \\ \langle y g_n(\vec{x})\rangle \end{pmatrix}$$

- solve $M\vec{p} = \vec{r}$ to get parameters in minimum
  - covariance matrix of track parameters $\vec{p}$ is $M^{-1}$
  - next slide: how to solve...

# solving linear systems

- solving $M\vec{p} = \vec{r}$ on a computer is done with *matrix decomposition*
  - strategy: write as product: $M = AB$ where $A$ is easily invertible, and $B$ allows for solution through substitution
  - example: $QR$ decomposition ($M = QR$)
    - $Q$: rotation/mirror matrix ($QQ^T = 1 = Q^TQ$)
    - $R = \begin{pmatrix} * & * & \dots & * \\ 0 & * & & \vdots \\ 0 & 0 & \ddots & \vdots \\ 0 & \dots & 0 & * \end{pmatrix}$
    - $M\vec{x} = \vec{r}$ would be solved as $\vec{w} = Q^T\vec{x}$ and $R\vec{x} = \vec{w}$
- solving is both faster and more accurate than inverting $M$, and using $\vec{x} = M^{-1}\vec{r}$

  → Solve, don't invert (unless you really need the inverse of $M$)
- many types of decomposition available
  - $LU$, $QR$, Cholesky, SVD, ...
  - how to choose?

# numerical stability of matrix decompositions

- decomposition $M = AB$ – how accurate can this be?
  - floating point isn't $\mathbb{R}$, you always have roundoff errors
  - we apply transformation $A^{-1}$ from left to get $B$ and right hand side
    - let's say $A^{-1}$ has eigenvalues $|\lambda_0| \le |\lambda_1| \le \ldots \le |\lambda_n|$
  - how does $A^{-1}$ act on roundoff? does it amplify?
    - overall scaling of numerical roundoff does not matter
  - *condition number* $\kappa = \frac{|\lambda_n|}{|\lambda_0|} = \frac{|\lambda_{max}|}{|\lambda_{min}|}$ matters (i.e. amplification contrast along eigenvectors)
- numerically stable decomposition schemes have $\kappa = 1$:
  - $QR$ decomposition: general invertible matrix
  - $LDL^T$ decomposition: symmetric invertible matrix
  - Cholesky decomposition: symmetric matrix with only positive EVs
  - SVD: if your matrix is problematic, or not even invertible (read up on it in a good book)
- stay clear of $LU$ decomposition if you value your result: $\kappa$ can easily be $10^4 \ldots 10^6$, depending on your $M$

# Cholesky decomposition

- Cholesky decomposition: for symmetric positive definite matrices $M = M^T > 0$
  - remember, we're looking for a minimum in $\chi^2$ – if you move out of it, $\chi^2$ increases, so $M$ must have only positive EVs
- decompose $M = LL^T$ with $L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \ddots & \dots \\ \vdots & * & \ddots & 0 \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}$
- consider related $M = \bar{L}D\bar{L}^T$ with $D = diag(l_{11}^2, \dots, l_{nn}^2)$ and $\bar{l}_{ij} = \frac{l_{ij}}{l_{ii}}$
  - EVs are zeroes of characteristic polynomial $det|\bar{L} - \lambda 1|$
  - they're all 1 → $\kappa = 1$
- Cholesky decomposition is numerically stable

# Cholesky decomposition

- Cholesky decomposition can use packed matrix storage – only save the diagonal and below (blue, in reading order):
  - $$M = \begin{pmatrix} m_{00} & m_{10} & \dots & m_{n0} \\ m_{10} & m_{11} & \dots & m_{n1} \\ \vdots & * & \ddots & \vdots \\ m_{n0} & m_{n1} & \dots & m_{nn} \end{pmatrix}$$
  - only need to update about half the amount of memory when adding measurements to the fit
  - for e.g. a 64 parameter fit, that's reading and writing about 16 kiB of RAM instead of 32 kiB for *each measurement*
- how to use this in your code?

```
#include <Math/CholeskyDecomp.h> // from ROOT
// use packed matrix storage { m00, m10, m11, ...}
// matrix element ij can be found at index (i * (i + 1)) / 2 + j
std::vector<double> mat = get_packed_mat();
std::vector<double> rhs = get_rhs();
unsigned nparams = rhs.size();
CholeskyDecompGenDim<double> decomp(n, mat.data());
if (!decomp) throw std::runtime_error("matrix not positive definite");
decomp.Solve(rhs);
// rhs now contains the solution p of mat * p = rhs
// if you need the covariance:
// decomp.Invert(mat.data()); // mat now contains covariance
```

# track parameter correlation studies (Hasret Nur)

- What can we do with this fitting framework?
- Hasret will study track models in the Mighty Tracker
- in the past, used this model in main tracker[1]:

$$dz = z - z_0$$
$$x(dz) = (((1 + dRatio \cdot dz) \cdot c)dz + b)dz + a$$
$$y(dz) = b' \cdot dz + a'$$

- idea: fit MCHits of (MC) particles in $xz$ and $yz$ projection with polynomials
- dump resulting parameters to tuple, study correlations
- with the excellent resolution of a pixel tracker, model above may no longer be sufficient
- Hasret will study what the best parametrisation is

[1] please evaluate polynomials like this (Horner's scheme) — very CPU efficient, and many CPUs have a specialized instruction fma(b, dz, a)=$b \cdot dz + a$ (fused multiply add).

# Chebyshev polynomials

- definition: $(x \in [-1, 1])$

$$T_0(x) = 1 \qquad T_1(x) = x \qquad T_n = 2xT_{n-1}(x) - T_{n-2}(x)$$

or

$$T_n(x) = \cos(n \arccos(x))$$

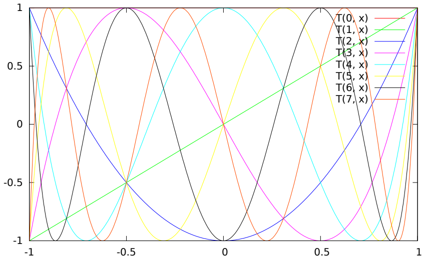
- fast: for fixed x, can evaluate with 1 or 2 floating point operations per order n
- these are orthogonal:

$$\int_{-1}^{1} T_j(x) T_k(x) \frac{dx}{\sqrt{1-x^2}} = \begin{cases} 0 & j \neq k \\ \frac{\pi}{2} & j = k \neq 0 \\ \pi & j = k = 0 \end{cases}$$

or

$$\sum_{l=0}^{N} T_j(x_l) T_k(x_l) = \begin{cases} 0 & j \neq k \\ N & j = k = 0 \\ \frac{N}{2} & j = k \neq 0 \end{cases} \qquad (x_l = \cos(\frac{l\pi}{N}))$$

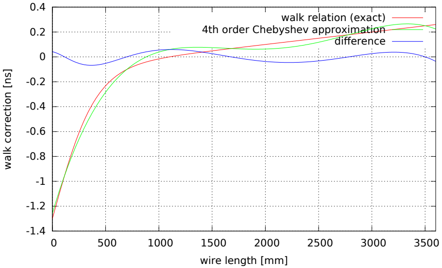
# Chebyshev polynomials




- approximate $f(x) \approx \sum_{k=0}^{n} c_k T_k(x)$
- Chebyshev polynomials intimately related with Fourier transforms
  - → fast convergence for well behaved functions
- best of all: error estimates are easy: $|T_k(x)| \le 1$
  - → accurate error estimate from summing up first few neglected $|c_k|$

# example: Chebyshev-expanded OT walk relation

- need to correct for time walk in OT, depends on length $l$ of hit along anode wire
- parameters for walk correction come from conditions DB
- calculate Chebyshev expansion on the fly in run 1/2 software
- can truncate after $c_4$ (e.g. with parameter's from Alexandr Koslinsky's thesis): $c_0 = -0.116724, c_1 = 0.544860, c_2 = -0.290254, c_3 = 0.196250, c_4 = -0.110068$




est. error: 0.0788 ns (scanning shows max. deviation $< 0.0676$ ns)

- notice how well-behaved the approximation error is ($|T_k(x)| \leq 1$)

# track propagator approximations

- let's put the pieces together
- write tuple: generate state vectors $(x, y, tx, ty, q/p)$ at fixed $z$
  thanks Renato for the tuples and code
- propagate through magnetic field to different $z$ locations
- use approximate symmetry of LHCb to fit only one quadrant:
  - if $x_T < 0$: $x \to -x$, $tx \to -tx$, $q/p \to -q/p$
  - if $y_T < 0$: $y \to -y$, $ty \to -ty$
- fit $p \in x, y, tx, ty$ with multi-dimensional Chebyshev series, e.g.

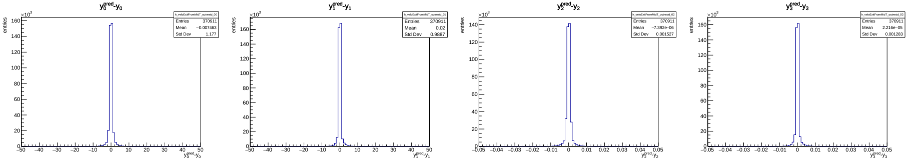
$$p_{VeloExit} = \sum_{i,j,k,l,m} p_{ijklm} T_i(x_T) T_j(y_T) T_k(tx_T) T_l(ty_T) T_m((q/p)_T)$$

  - $p_{ijklm}$ are the fit parameters
  - I am suppressing the linear transformation that brings the track parameter ranges to the $[-1, +1]$ interval for the Chebyshev polynomials

# tuple number 1

- all initial states at the origin, flat distribution in tx and ty (100 steps from -0.4 to 0.4)
- flat in q/p (200 steps from 1/(100 GeV) to 1/(500 MeV)), both charges
- then propagate through magnetic field to these values in z ([mm]):
  - 770 (VeloExit)
  - 2307, 2313, 2322, 2328, 2362, 2368, 2377, 2383, 2586, 2592, 2601, 2608, 2641, 2647, 2656, 2663 (UT layers, last one UTExit)
  - 5240 (somewhere near the middle of the magnet)
  - 7500, 8520, 9410 (BeginT, MidT, EndT)
- 1.6 M propagated states (many states do not reach T)
- idea here is to focus on essentially prompt tracks – the q/p estimate we use to get the q/p for a T track segment has that assumption built in anyway

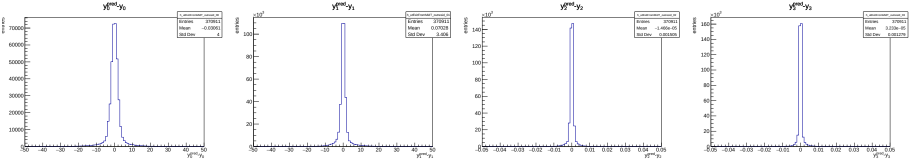
# from $z_{MidT}$ to $z_{VeloExit}$



$x$ $y$ $tx$ $ty$

- for $p > 3$ GeV, get RMS of 1.2 mm/1.0 mm/$1.5 \cdot 10^{-3}/1.3 \cdot 10^{-3}$ in $x/y/tx/ty$
  - not perfect, but real tracks have multiple scattering – likely good enough…
- include Chebyshev up to first order in $x$, $y$, $tx$, $ty$, and up to fifth order in $q/p$
- 2 * 2 * 2 * 2 * 6 = 96 fit parameters
- can do 16 fits w. 96 parameters on 1.6 M tracks in less than a neutron lifetime
- can evaluate at throughput greater than 1.5 Mtracks/s on single core of 13 year-old laptop

# from $z_{MidT}$ to $z_{UTExit}$



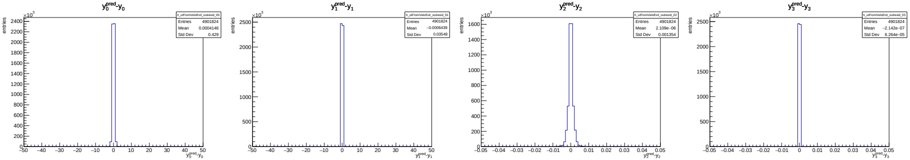

$x$ $y$ $tx$ $ty$

- for $p > 3$ GeV, get RMS of 4.0 mm/3.4 mm/$1.5 \cdot 10^{-3}/1.3 \cdot 10^{-3}$ in $x/y/tx/ty$
  - not perfect, but real tracks have multiple scattering – likely good enough…
- include Chebyshev up to first order in $x$, $y$, $tx$, $ty$, and up to fifth order in $q/p$
- 2 * 2 * 2 * 2 * 6 = 96 fit parameters
- less good than the Velo one on the last page (pointing constraint weaker, $\vec{B}$ field starts to act!)

# tuple number 2

- all initial states at $z_{EndT}$
- use N = 50
- try out Chebyshev-based spacing of points:
  $\frac{max+min}{2} + \frac{max-min}{2}\cos(\pi\frac{2k+1}{2N})$ for $k = 0, \dots, N-1$
  - x: min = 0 mm, max = 3200 mm; add mirror to add the other half of detector
  - y: min = 0 mm, max = 2800 mm; add mirror to add the other half of detector
  - tx: min = -0.8, max = 0.8
  - ty: min = -0.4, max = 0.4
  - q/p: min = -0.002, max = 0.002
- propagate to same $z$ values as last tuple
- idea here was to optimize for potentially non-prompt tracks, if we manage to pull out the correct pair of Velo segment and T segment

# from $z_{VeloExit}$ into UT




$x$ $y$ $tx$ $ty$

- parametrise in $(x, y, tx, ty, q/p, z_{UT})^T$
- for $p > 3$ GeV, get RMS of 0.43 mm/0.35 mm/$1.4 \cdot 10^{-3}/6.3 \cdot 10^{-5}$ in $x/y/tx/ty$
- include Chebyshev up to first order in $x$, $y$, $tx$, $ty$, $z_{UT}$, and up to second order in $q/p$
- 2 * 2 * 2 * 2 * 3 * 2 = 96 fit parameters

# summary

- approximations can propagate a state vector through magnetic field
  - fairly low-order approximations can do a reasonable job predicting positions and slopes
  - they do so with relatively little CPU
- idea: use approximations to match tracks at the end of Velo, and find hits in UT
  - a KD tree is the data structure to use (see Arthur's talk or backup)
    - finds nearby tracks in parameter space
    - like std::sort and search windows, but in more then 1 dimension
    - needs $\mathcal{O}(N \log N)$ work to build the tree, and $\mathcal{O}(\log N)$ work to get nearest neighbour(s) in parameter space
    - could also be useful to find hits nearby in position and time (timing subdetectors?)
  - Renato will report on how well the matching works

# summary

- open questions
  - I am not at all sure these are the optimal parametrisations
  - up to which orders?
  - which granularity? (approximating a whole quadrant is maybe a bit crazy)
  - how to best generate the tuples for fitting (best distribution in track state space for fitting)
  - could imagine that, one day, we use such parametrisations to propagate all tracks (instead of referring to the field map)...

- fits linear in track parameters are fairly powerful
  - I hope it's your new secret superpower!
  - not only useful for pattern reco problems
  - probably should be used far more widely
- code is on gitlab (fitter code < 1700 lines of C++ incl. comments!)
  - feel free to take a look, maybe learn from, and to play with approximations

# backup

backup

# recap: efficient hit finding

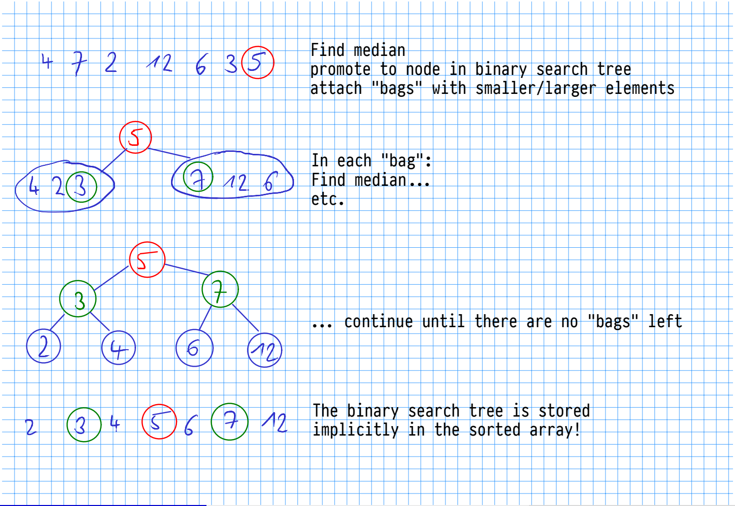
- we're all familiar with the `std::sort`/`std::lower_bound` combo:

```
auto firstHit = hitsInLayer.begin(), lastHit = hitsInLayer.end();
std::sort(firstHit, lastHit, [] (auto xa, auto xb) { return xa < xb; });
for (const auto& s: seeds) {
    const auto dz = layerZ - s.z();
    // predict coordinate in new layer
    const auto x = s.x() + x.xSlope() * dz;
    // open up a search window
    const auto xcov = s.covX() + dz * (2. * s.xCov() * s.xSlCov() + dz * s.xSlCov());
    const auto xerr = std::sqrt(xcov);
    const auto xmin = x - 3. * xerr, xmax = x + 3. * xerr;
    // loop over corresponding hits in region of interest
    for (auto it = std::lower_bound(firstHit, lastHit, xmin,
            [] (auto xa, auto xb) { return xa < xb; });
            lastHit != it && it->x() < xmax; ++it) {
        const auto& h = *it;
        // do something to seed s and compatible hit h
    }
}
```

- well known technique, $O(N \log N)$ work for `sort`, $O(\log N)$ work for `lower_bound`
- great for cases where we have a single coordinate, not so good in two/more dimensions
- can we generalise?

# recap: binary search trees

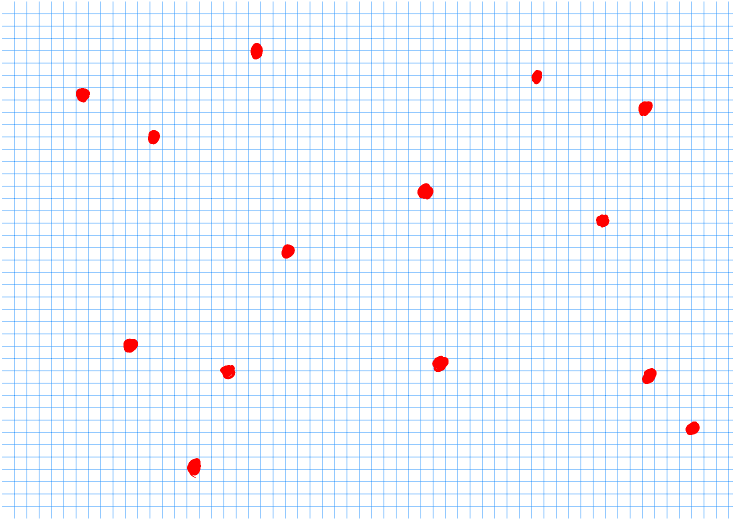
The `std::sort`/`std::lower_bound` trick works because it builds a balanced binary search tree…



# kd trees

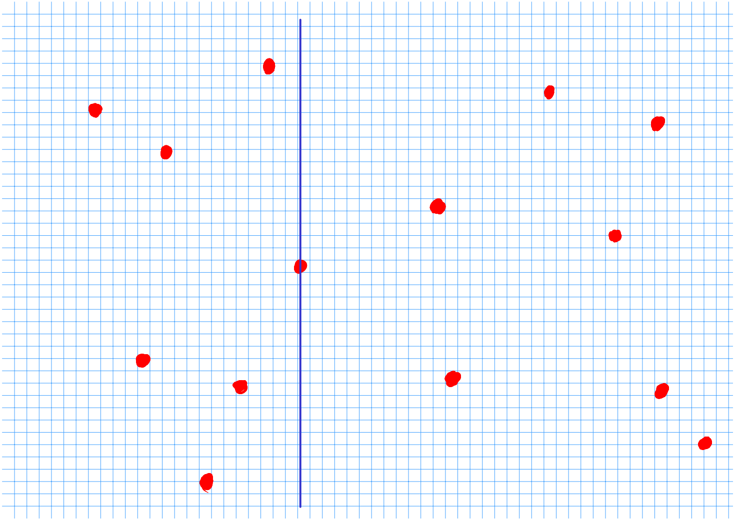
- a kd tree is a straightforward extension of that idea
  - recursively pick median of “bag” or sub-array as in example above
  - cycle through the dimensions
  - searching is based on the same idea as `lower_bound`, but sometimes needs to check the other subtree on its way up towards the root, as there is more than one dimension
- wait, can we have an example?

# kd trees: building a 2d tree (1/4)



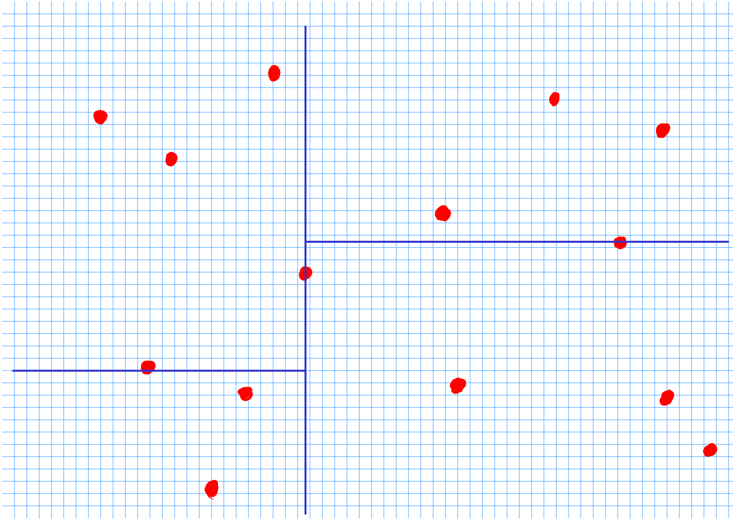
start with some points in 2D...

# kd trees: building a 2d tree (2/4)



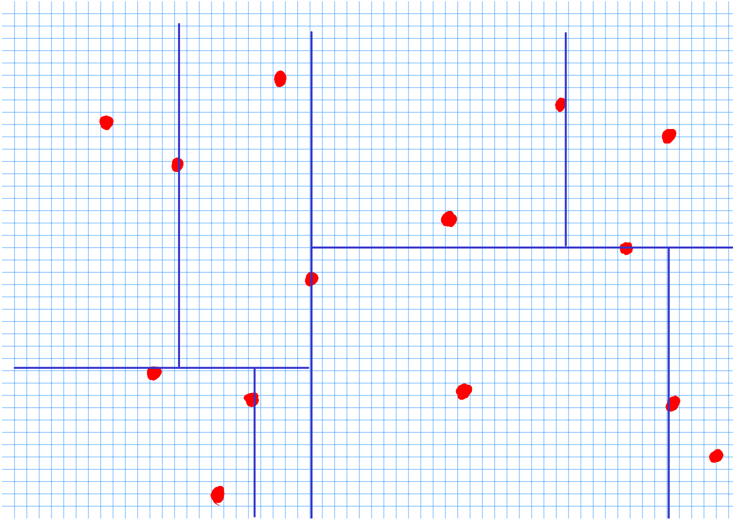
...find median along one axis, promote to tree node...

# kd trees: building a 2d tree (3/4)



…find median in subsets along next axis (cyclically), promote to tree

# kd trees: building a 2d tree (4/4)



…and continue until the whole tree is built.

# kd trees: code example (1/2)

- whereas the `std::sort`/`std::lower_bound` trick only needed the comparison functor, kd trees need
  - comparison of items along named axis ($x/y/\dots$)
  - distance functor (or monotonic function of distance)
- code example using single-header package kdtree

```
using point = std::array<float, 2>; // 2D points
const auto cmp = [] (const point& a, const point& b, auto dim) noexcept {
    return a[dim] < b[dim]; };
const auto dist = [] (const point& a, const point& b, auto dim) noexcept {
        if (std::size_t(-1) == dim) { // full distance
            // a point is not it's own nearest neighbour (depends
            // on application if you want this...)
            if (&a == &b) return std::numeric_limits<float>::max();
            // full distance between points - we use squared distance
            // here to save a square root
            return (a[0] - b[0]) * (a[0] - b[0]) +
                (a[1] - b[1]) * (a[1] - b[1]);
        } else { // distance in coordinate dim only
            return (a[dim] - b[dim]) * (a[dim] - b[dim]);
        }
    };
```

- with these two helper functions, we can now find nearest neighbours in k dimensions...

# kd trees: code example (2/2)

- we can now build a kd tree, and find point closest to a given point:

```
// okay, get some points from somewhere:
using Points = std::vector<point>;
Points v = /* from somewhere ... */;
// build kd tree
build_kdtree<2>(v.begin(), v.end(), cmp);
// find nearest neighbour to a point
const auto& p = *(v.begin() + 42); // some element - need not be one from v
auto nearest = find_nearest_kdtree<2>(v.begin(), v.end(), p, cmp, dist);
std::cout << "nearest is (" << (*nearest)[0] << "," << (*nearest)[1] << ")"
    << std::endl;
```

- can also find more than one nearest neighbour:

```
// pair of iterator (to neighbour), and its distance to a point
using NeighbourWithDistance = std::pair<Points::iterator, float>;
// array of five of these
using FiveBest = std::array<NeighbourWithDistance, 5>;

// prepare an array, fill with "nothing found"
FiveBest best;
best.fill(std::make_pair(v.end(), std::numeric_limits<float>::max()));
// find the five nearest neighbours to p
find_n_nearest_kdtree<2>(v.begin(), v.end(), p, best.begin(), best.end(),
    cmp, dist);
for (const auto& n: best) {
    std::cout << "near neighbour is (" << (*n.first)[0] << ","
        << (*n.first)[1] << ") dist" << n.second << std::endl;
}
```

# conclusion

- kd trees allow
  - $O(\log N)$ searching for nearest neighbour(s) in k dimensions
  - need $O(N \log N)$ work to build kd tree initially
- if you want to play: a simple C++ version is available in the kdtree package
- possible areas of application
  - building block for tracking in pixel detector
  - matching tracks based on track parameters
  - tracking in detectors that supply hit time information
  - …(your idea here)
- I hope to get people thinking, and am willing to answer questions, and help, but probably won't have time to work on something myself