EVALUATING GEOMAGNETIC CUTOFFS WITH HIGH-PERFORMANCE 3-D TRAJECTORY CALCULATIONS

Keito Watanabe

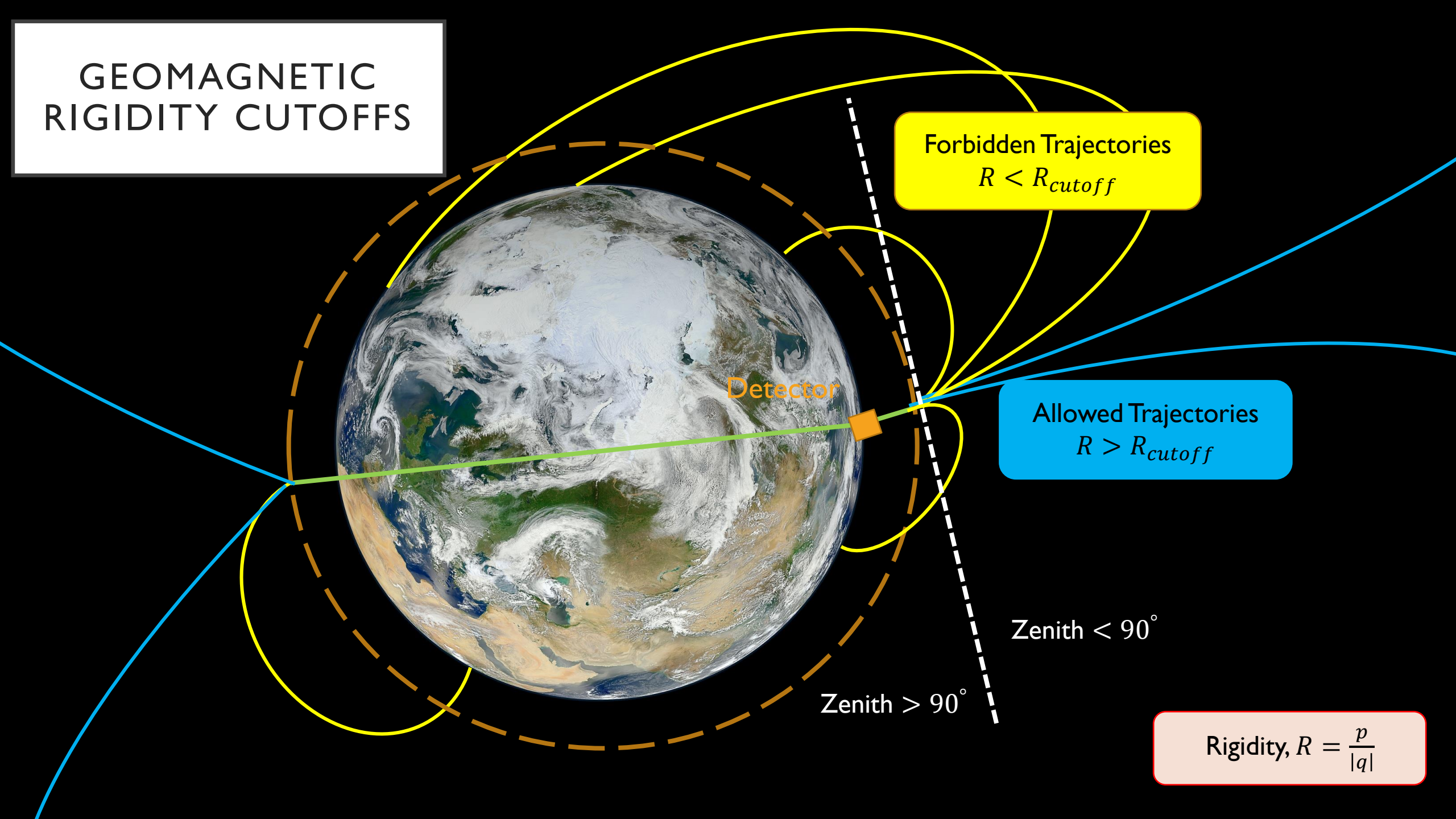
University of Alberta

October 9th, 2020

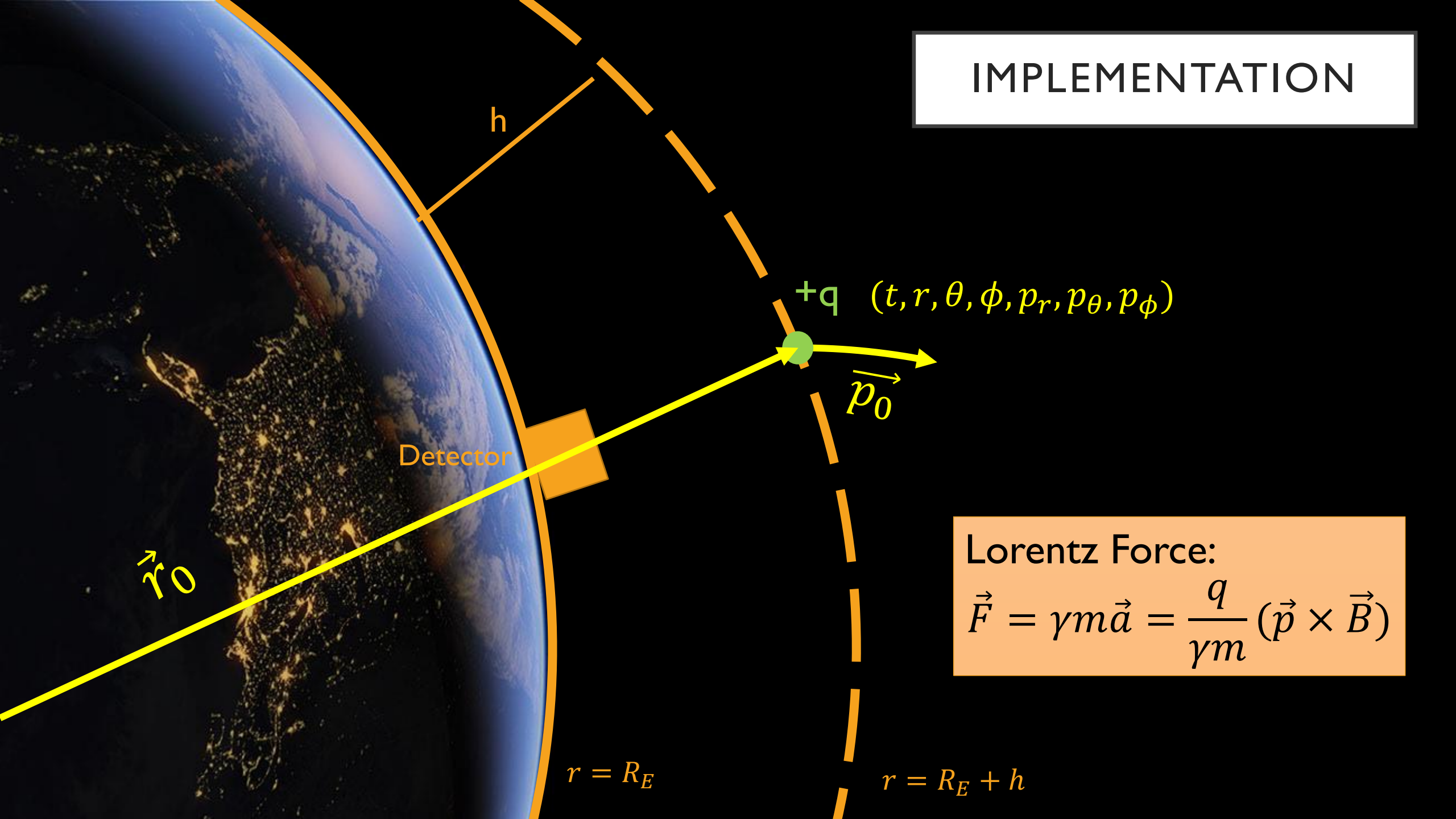
GOALS

- Simulate cosmic ray trajectories within the geomagnetic field
- Perform optimization & parallelisation
- Evaluate the geomagnetic rigidity cutoffs for 3D neutrino flux calculations

GEOMAGNETIC RIGIDITY CUTOFFS



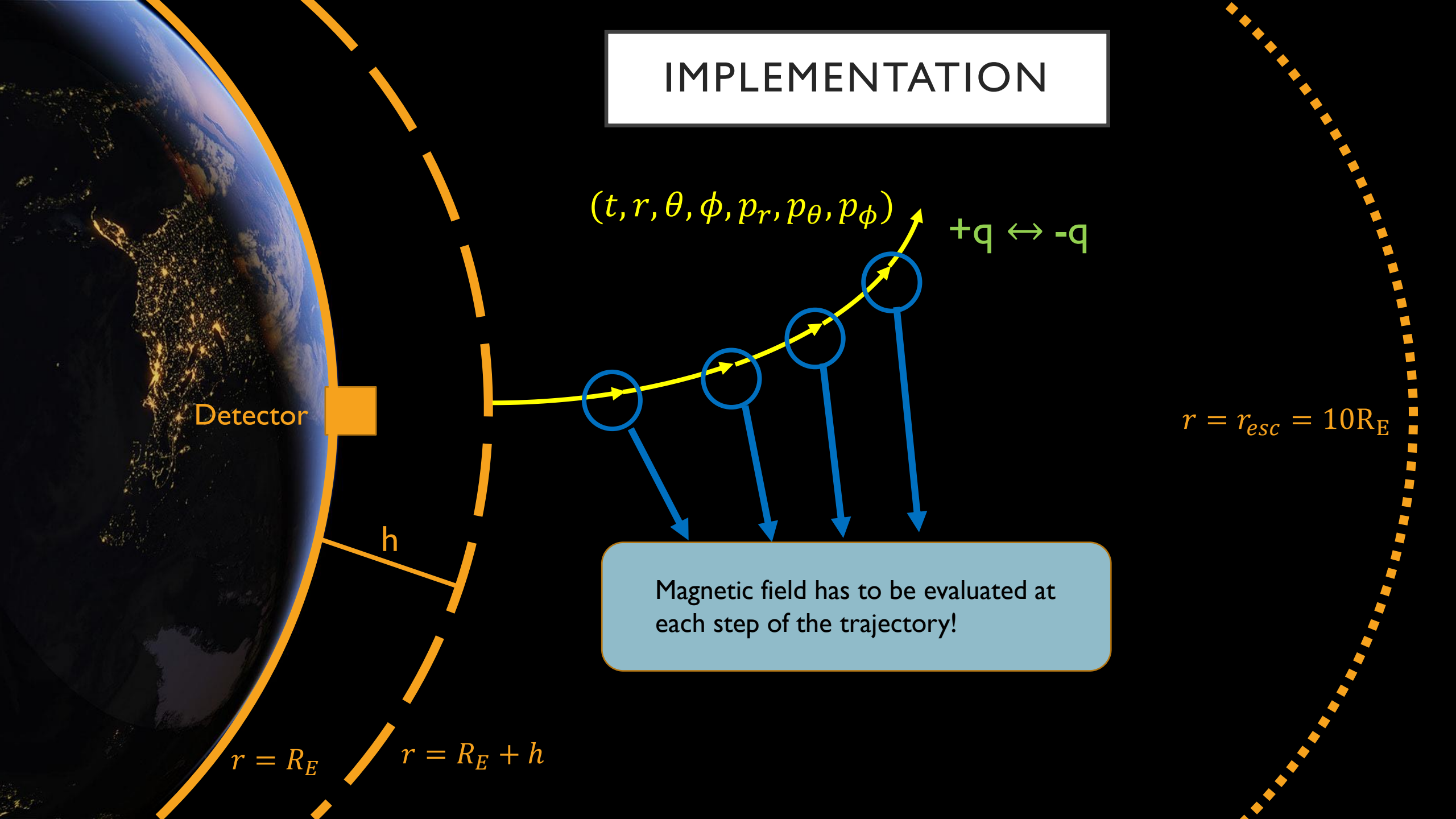
IMPLEMENTATION



Lorentz Force:

$$\vec{F} = \gamma m \vec{a} = \frac{q}{\gamma m}(\vec{p} \times \vec{B})$$

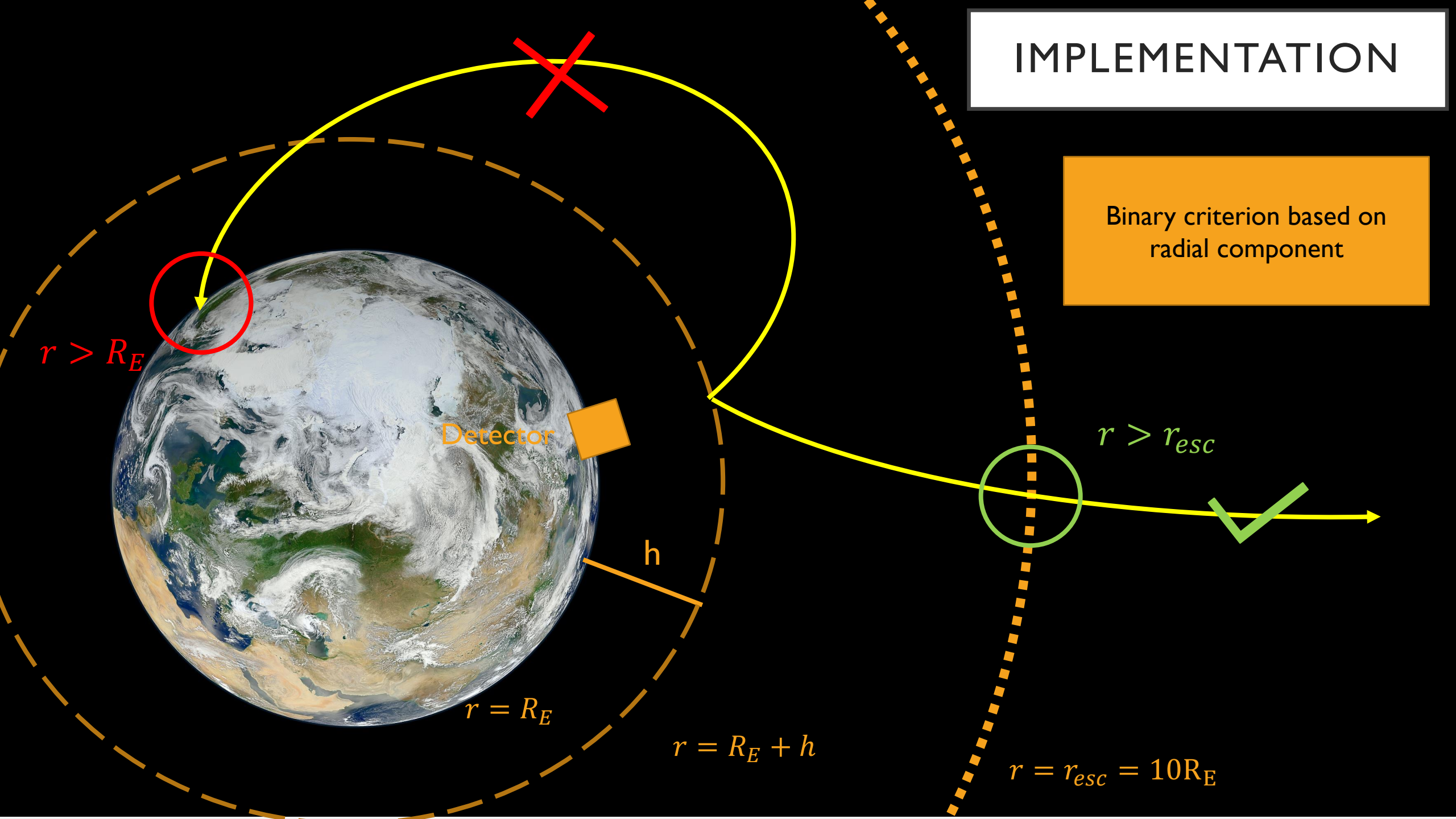
IMPLEMENTATION



Magnetic field has to be evaluated at each step of the trajectory!

IMPLEMENTATION

Binary criterion based on radial component



4TH-ORDER RUNGE-KUTTA

- Truncation error of $O(h^5)$

  (cf. Euler $O(h^2)$)
- Widely used in literature and in space physics for electron trajectory calculations

Algorithm:

$$\vec{k}_1 = hf(t_n, \vec{s}_n)$$

$$\overrightarrow{k_2} = hf\left(t_n + \frac{1}{2}h, \overrightarrow{s_n} + \frac{1}{2}\overrightarrow{k_1}\right)$$

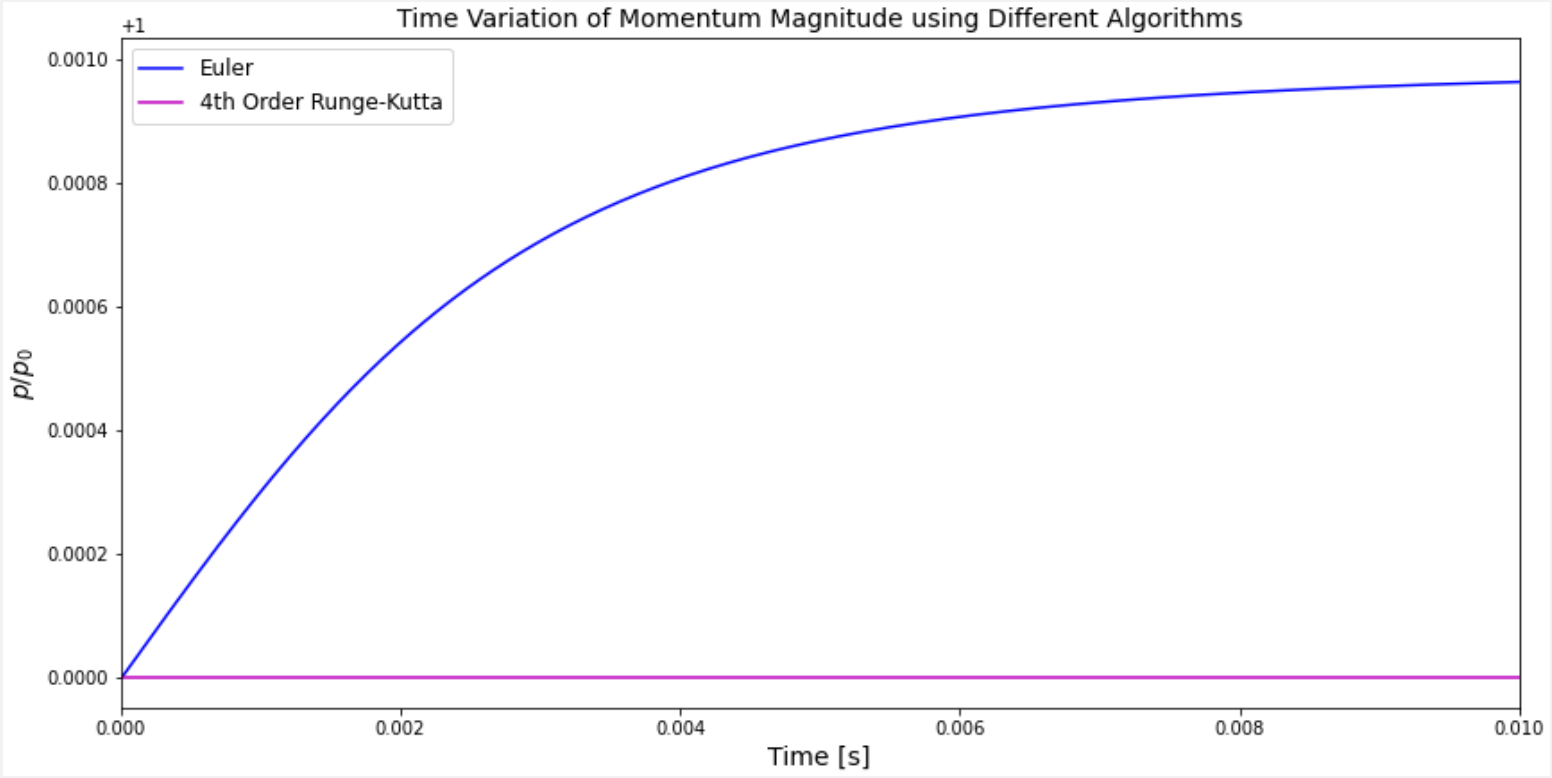
$$\overrightarrow{k_3} = hf\left(t_n + \frac{1}{2}h, \overrightarrow{s_n} + \frac{1}{2}\overrightarrow{k_2}\right)$$

$$\vec{k}_4 = hf(t_n + h, \vec{s}_n + \vec{k}_3)$$

$$\vec{k} = \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)$$

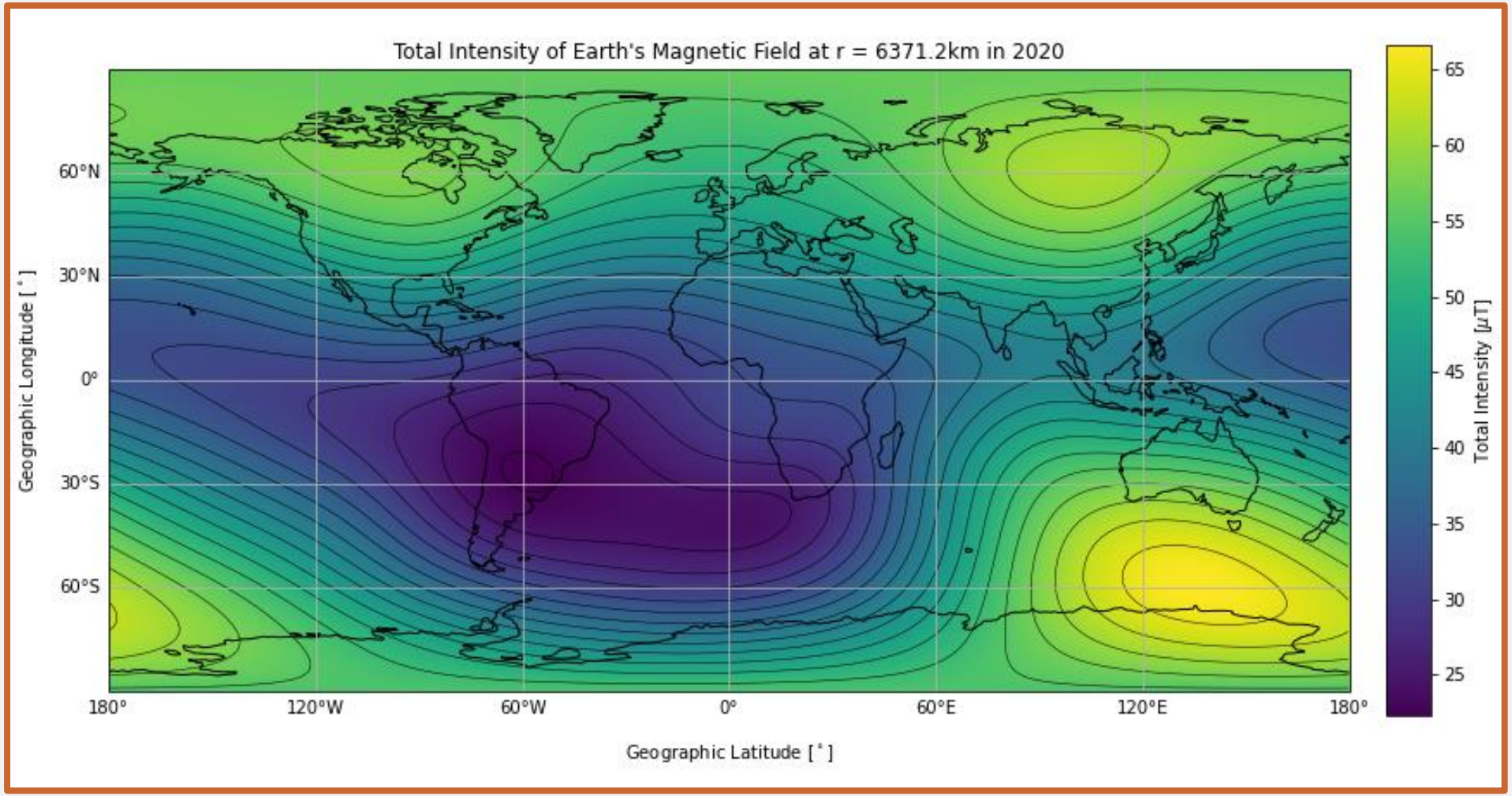
$$\vec{s}_{n+1} = \vec{s}_n + \vec{k}$$

$$t_{n+1} = t_n + h$$



IGRF MODEL

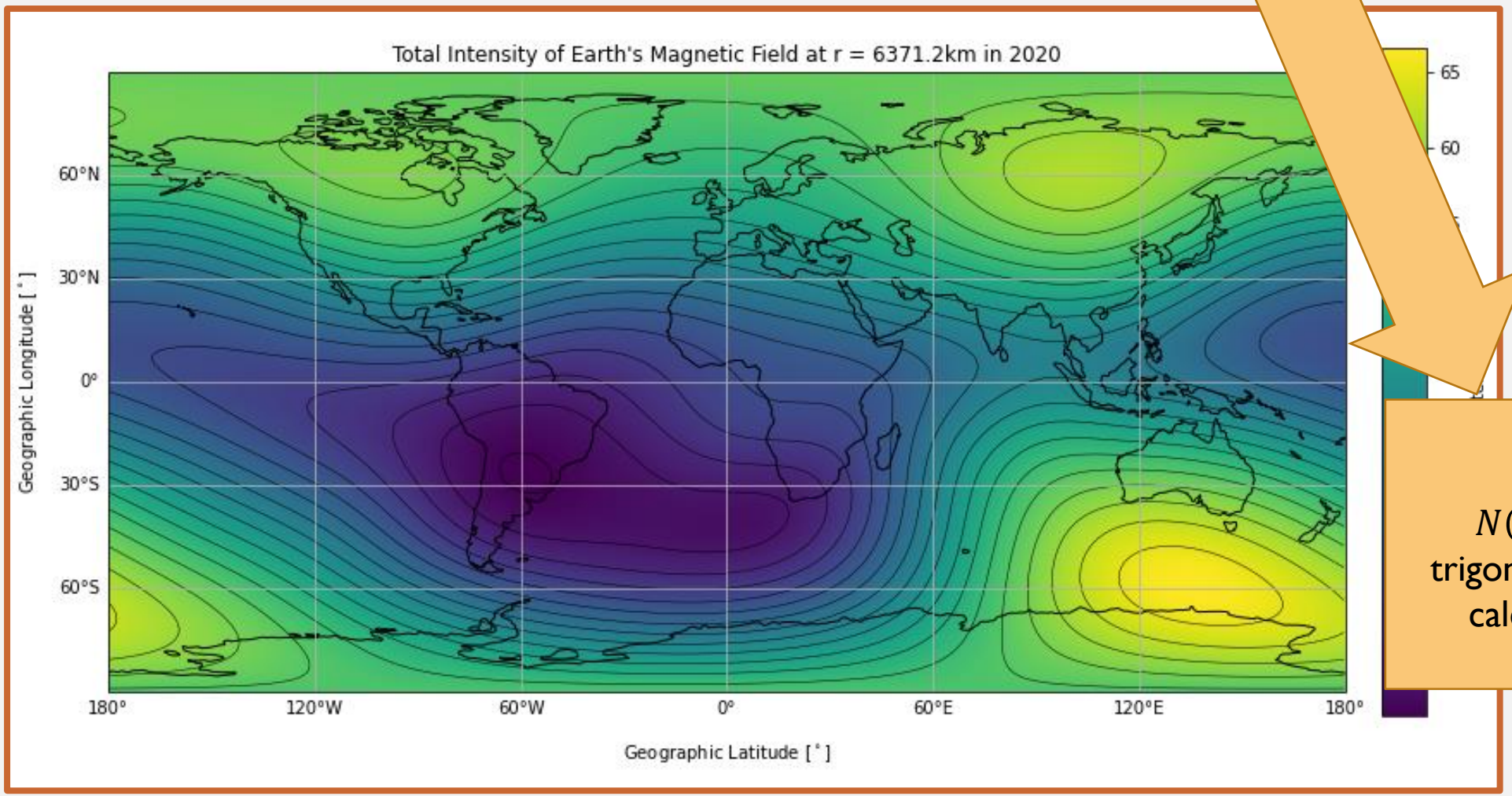
$$V(r,\theta,\phi) = R_E \sum_{n=1}^{N} \sum_{m=0}^{n} \left(\frac{R_E}{r}\right)^{n+1} [g_n^m(t)\cos(m\phi) + h_n^m(t)\sin(m\phi)] P_n^m(\cos\theta)$$



https://www.ngdc.noaa.gov/IAGA/vmod/igrf.html

IGRF MODEL

$$V(r,\theta,\phi) = R_E \sum_{n=1}^{N} \sum_{m=0}^{n} \left(\frac{R_E}{r}\right)^{n+1} [g_n^m(t)\cos(m\phi) + h_n^m(t)\sin(m\phi)] P_n^m(\cos\theta)$$



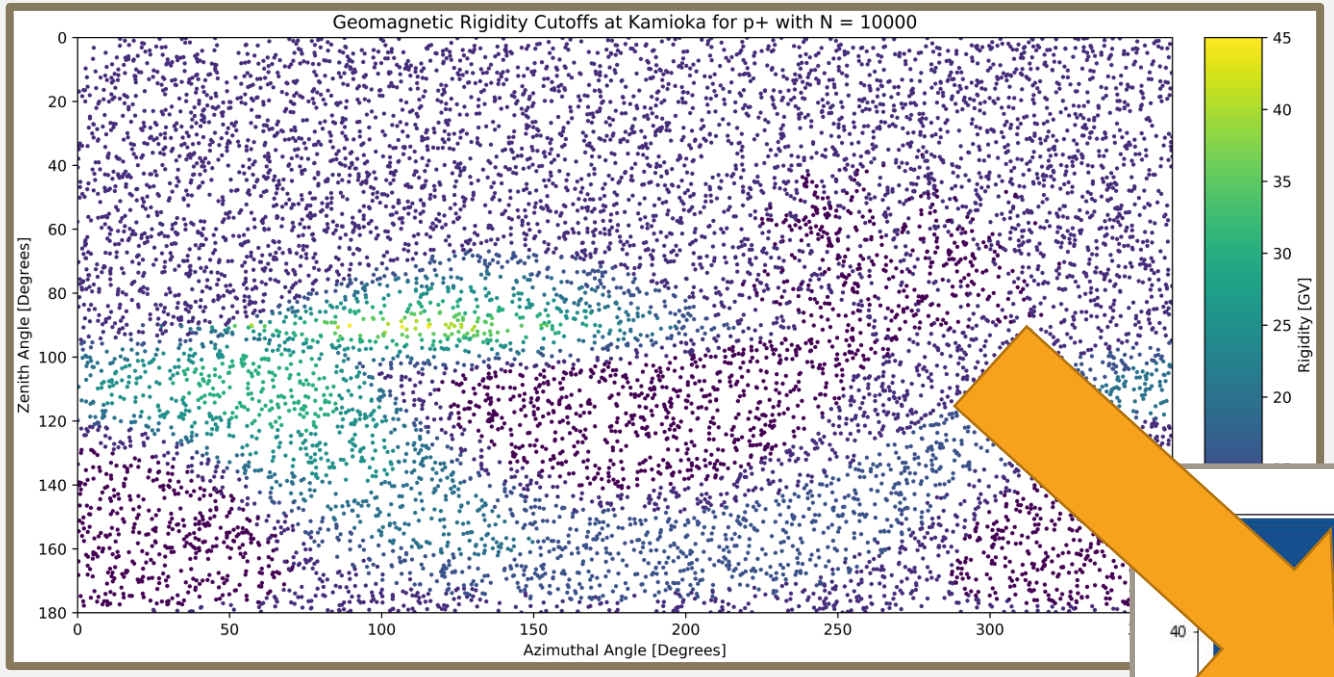
Total of $N(N+3) \approx 208$ trigonometric terms to calculate per step!

(N=13 as of 2020)

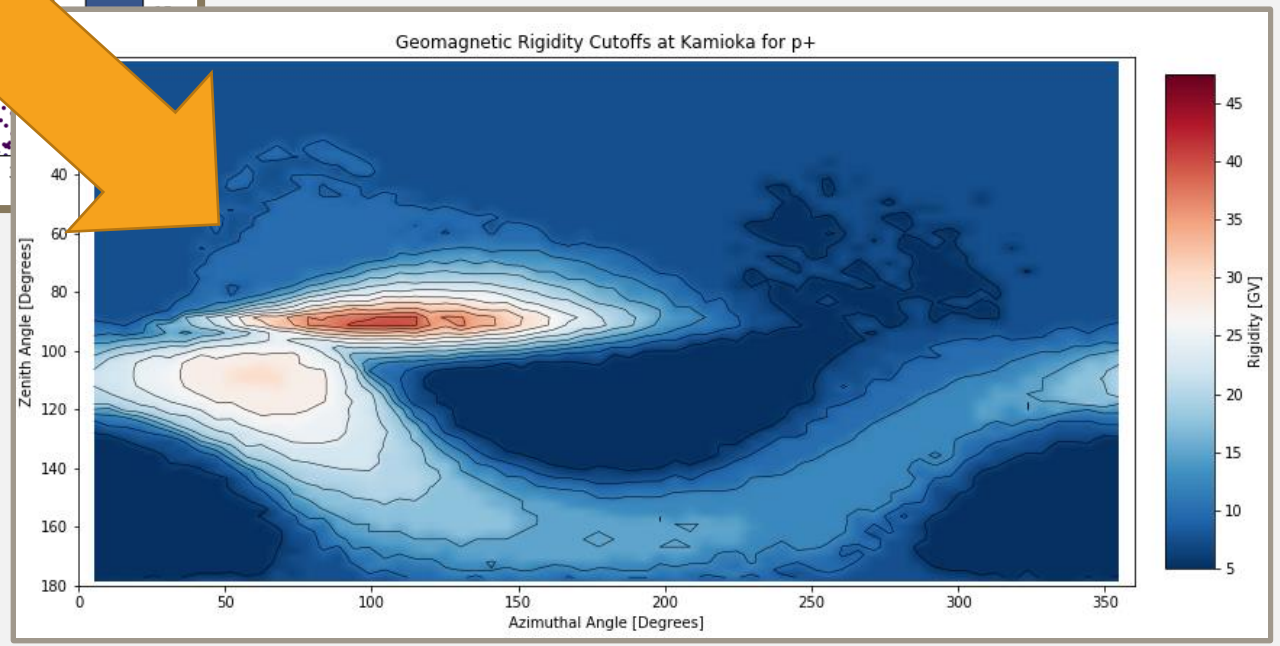
https://www.ngdc.noaa.gov/IAGA/vmod/igrf.html

ALGORITHM: RIGIDITY CUTOFF

Different R_{cutoff} based on zenith and azimuthal direction with respect to detector frame



MC Sampling



Linearly Interpolated Results

OPTIMIZATION: PYTHON VS C++

Performance with C++ is $\sim 10^3$ better than Python version!

Python:

- Slow loop evaluation
- Function call overhead

C++:

- Minimal function call overhead
- Code optimization performed in compile time
- More control for optimization procedures (const, pass-by-reference etc.)

Type	Iterations per Second	Performance Ratio
Python	0.286	1.51×10^{-3}
C++ (Scalar Form)	226.23	1.194
C++ (Vector Form)	189.35	1

VECTOR VS SCALAR RUNGE-KUTTA METHODS

Different forms of the Runge-Kutta algorithm affect the stability and performance of the code

Vector Form:

$$\vec{f} = \vec{f}(t, \vec{s});\ \vec{s}_{n+1} = \vec{s}_n + \vec{k};\ t_{n+1} = t_n + h$$

```
319
320   // start the loop
321   for (int i = 0; i < max_iter_; ++i) {
322     // evaluate the k-coefficients
323
324     std::array<double, 6> &k1_vec = h * ode_lrz(t, vec);
325     std::array<double, 6> &k2_vec =
326         h * ode_lrz(t + (0.5 * h), vec + (0.5 * k1_vec));
327     std::array<double, 6> &k3_vec =
328         h * ode_lrz(t + (0.5 * h), vec + (0.5 * k2_vec));
329     std::array<double, 6> &k4_vec = h * ode_lrz(t + h, vec + k3_vec);
330
331     std::array<double, 6> &k_vec =
332         (1. / 6.) * (k1_vec + (2. * k2_vec) + (2. * k3_vec) + k4_vec);
333
334     // increment by weighted sum
335     vec = vec + k_vec;
336     t += h;  // increase time
```

Scalar Form:

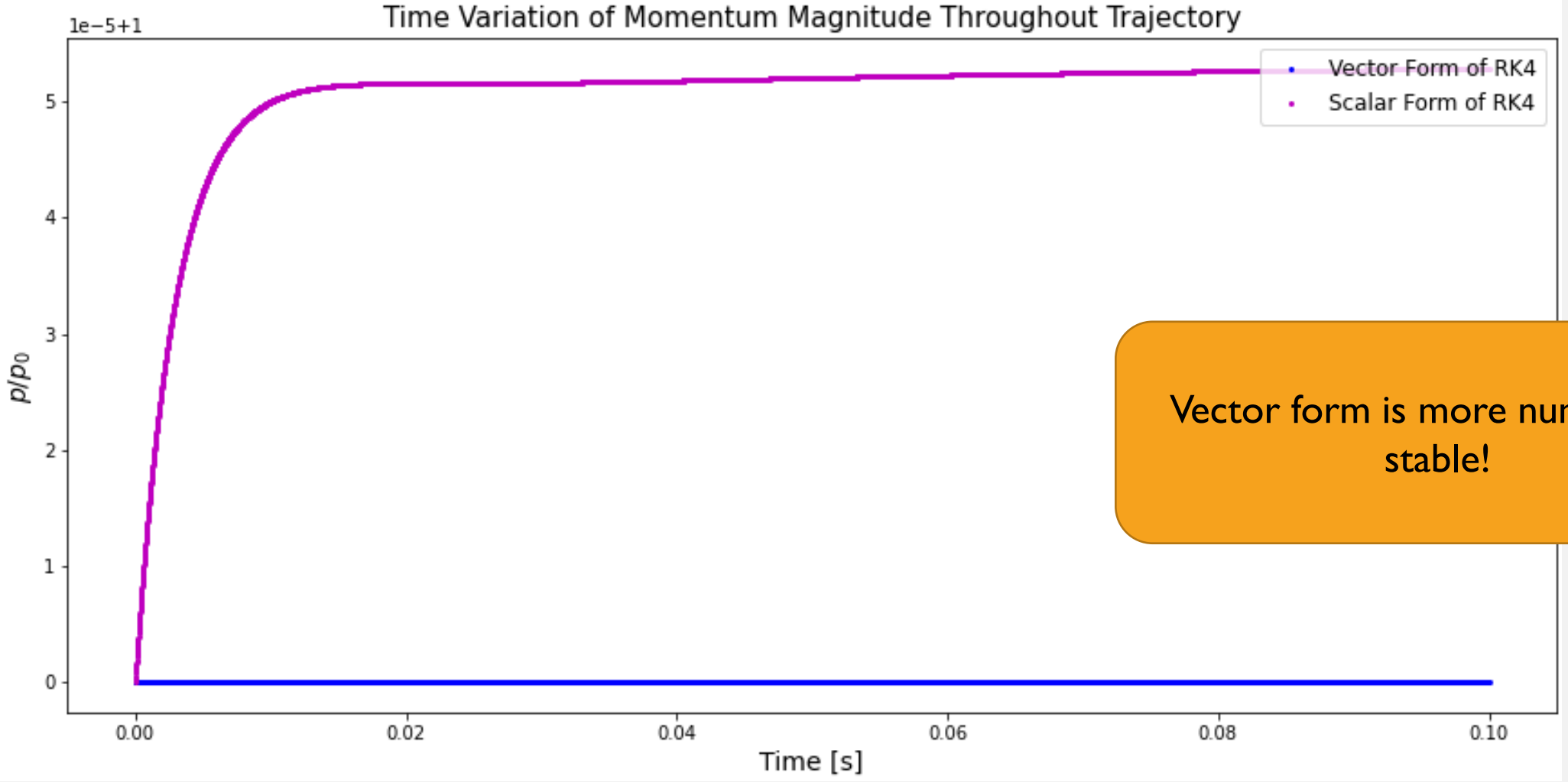
$$f_i = f_i(t, s_j)$$
$$s_{n+1_i} = s_{n_i} + k_i$$
$$t_{n+1} = t_n + h$$

$$\vec{s} = (\vec{r}, \vec{p}) = (r, \theta, \phi, p_r, p_\theta, p_\phi)$$

```
354   double r_k1 = stepsize_ * dr_dt(pr);
355   double theta_k1 = stepsize_ * dtheta_dt(r, ptheta);
356   double phi_k1 = stepsize_ * dphi_dt(r, theta, pphi);
357   double pr_k1 = stepsize_ * dpr_dt(r, theta, phi, pr, ptheta, pphi);
358   double ptheta_k1 = stepsize_ * dptheta_dt(r, theta, phi, pr, ptheta, pphi);
359   double pphi_k1 = stepsize_ * dpphi_dt(r, theta, phi, pr, ptheta, pphi);
360
361   double r_k2 = stepsize_ * dr_dt(pr + 0.5 * pr_k1);
362   double theta_k2 =
363       stepsize_ * dtheta_dt(r + 0.5 * r_k1, ptheta + 0.5 * ptheta_k1);
364   double phi_k2 = stepsize_ * dphi_dt(r + 0.5 * r_k1, theta + 0.5 * theta_k1,
365                                       pphi + 0.5 * pphi_k1);
366   double pr_k2 =
367       stepsize_ * dpr_dt(r + 0.5 * r_k1, theta + 0.5 * theta_k1,
368                          phi + 0.5 * phi_k1, pr + 0.5 * pr_k1,
369                          ptheta + 0.5 * ptheta_k1, pphi + 0.5 * pphi_k1);
370   double ptheta_k2 =
371       stepsize_ * dptheta_dt(r + 0.5 * r_k1, theta + 0.5 * theta_k1,
372                              phi + 0.5 * phi_k1, pr + 0.5 * pr_k1,
373                              ptheta + 0.5 * ptheta_k1, pphi + 0.5 * pphi_k1);
374   double pphi_k2 =
375       stepsize_ * dpphi_dt(r + 0.5 * r_k1, theta + 0.5 * theta_k1,
376                            phi + 0.5 * phi_k1, pr + 0.5 * pr_k1,
377                            ptheta + 0.5 * ptheta_k1, pphi + 0.5 * pphi_k1);
378
379   double r_k3 = stepsize_ * dr_dt(pr + 0.5 * pr_k2);
380   double theta_k3 =
381       stepsize_ * dtheta_dt(r + 0.5 * r_k2, ptheta + 0.5 * ptheta_k2);
382   double phi_k3 = stepsize_ * dphi_dt(r + 0.5 * r_k2, theta + 0.5 * theta_k2,
383                                       pphi + 0.5 * pphi_k2);
384   double pr_k3 =
385       stepsize_ * dpr_dt(r + 0.5 * r_k2, theta + 0.5 * theta_k2,
386                          phi + 0.5 * phi_k2, pr + 0.5 * pr_k2,
387                          ptheta + 0.5 * ptheta_k2, pphi + 0.5 * pphi_k2);
388   double ptheta_k3 =
389       stepsize_ * dptheta_dt(r + 0.5 * r_k2, theta + 0.5 * theta_k2,
390                              phi + 0.5 * phi_k2, pr + 0.5 * pr_k2,
391                              ptheta + 0.5 * ptheta_k2, pphi + 0.5 * pphi_k2);
392   double pphi_k3 =
393       stepsize_ * dpphi_dt(r + 0.5 * r_k2, theta + 0.5 * theta_k2,
394                            phi + 0.5 * phi_k2, pr + 0.5 * pr_k2,
395                            ptheta + 0.5 * ptheta_k2, pphi + 0.5 * pphi_k2);
396
397   double r_k4 = stepsize_ * dr_dt(pr + pr_k3);
398   double theta_k4 = stepsize_ * dtheta_dt(r + r_k3, ptheta + ptheta_k3);
399   double phi_k4 =
400       stepsize_ * dphi_dt(r + r_k3, theta + theta_k3, pphi + pphi_k3);
401   double pr_k4 =
402       stepsize_ * dpr_dt(r + r_k3, theta + theta_k3, phi + phi_k3, pr + pr_k3,
403                          ptheta + ptheta_k3, pphi + pphi_k3);
404   double ptheta_k4 =
405       stepsize_ * dptheta_dt(r + r_k3, theta + theta_k3, phi + phi_k3,
406                              pr + pr_k3, ptheta + ptheta_k3, pphi + pphi_k3);
407   double pphi_k4 =
408       stepsize_ * dpphi_dt(r + r_k3, theta + theta_k3, phi + phi_k3, pr + pr_k3,
409                            ptheta + ptheta_k3, pphi + pphi_k3);
410
411   traj_vector_.r +=
412       (1. / (6. * rel_mass)) * (r_k1 + 2. * r_k2 + 2. * r_k3 + r_k4);
413   traj_vector_.theta += (1. / (6. * rel_mass)) *
414                         (theta_k1 + 2. * theta_k2 + 2. * theta_k3 + theta_k4);
415   traj_vector_.phi +=
416       (1. / (6. * rel_mass)) * (phi_k1 + 2. * phi_k2 + 2. * phi_k3 + phi_k4);
417   traj_vector_.pr +=
418       (1. / (6. * rel_mass)) * (pr_k1 + 2. * pr_k2 + 2. * pr_k3 + pr_k4);
419   traj_vector_.ptheta += (1. / (6. * rel_mass)) * (ptheta_k1 + 2. * ptheta_k2 +
420                                                    2. * ptheta_k3 + ptheta_k4);
421   traj_vector_.pphi += (1. / (6. * rel_mass)) *
422                        (pphi_k1 + 2. * pphi_k2 + 2. * pphi_k3 + pphi_k4);
423   traj_vector_.t += stepsize_;
```

VECTOR VS SCALAR RUNGE-KUTTA METHODS

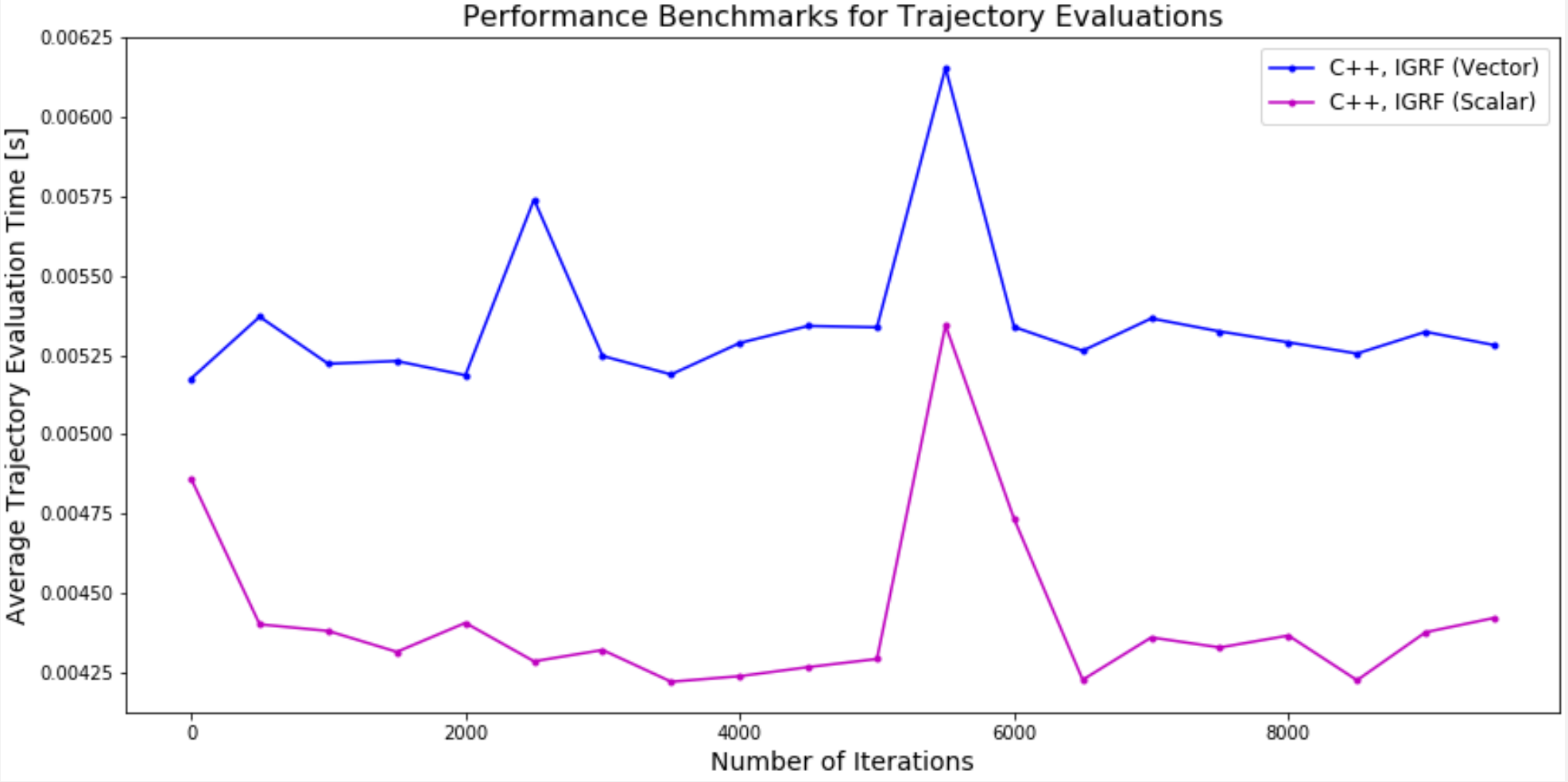
No energy loss term ⇒ $|\vec{p}|$ is constant throughout the trajectory



Vector form is more numerically stable!

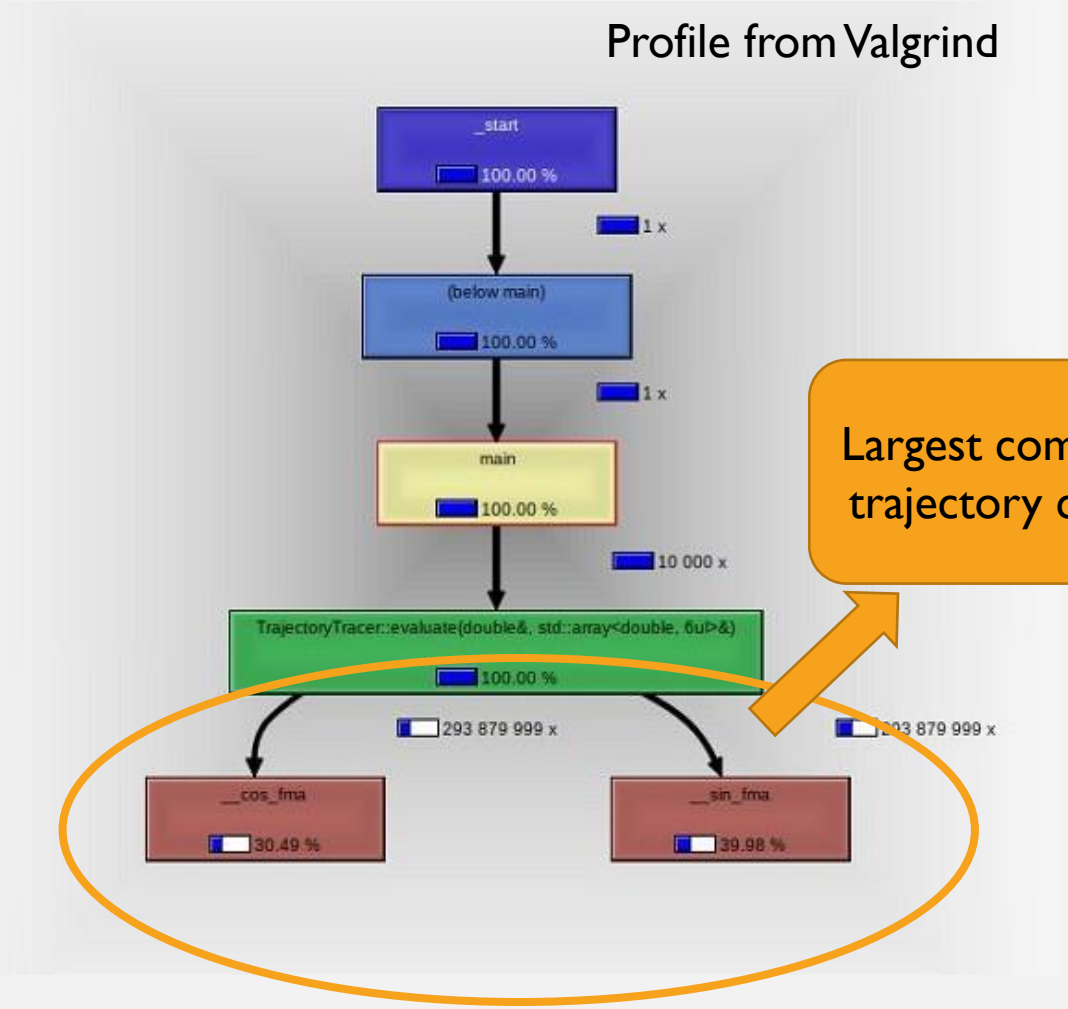
VECTOR VS SCALAR RUNGE-KUTTA METHODS

Vector form is ~0.8x slower than trajectory calculations in scalar form



PROFILING: TAN VS SIN/COS

Profile from Valgrind



Largest component of trajectory calculation

tan() is ~1.5x slower than $\frac{\sin()}{\cos()}$

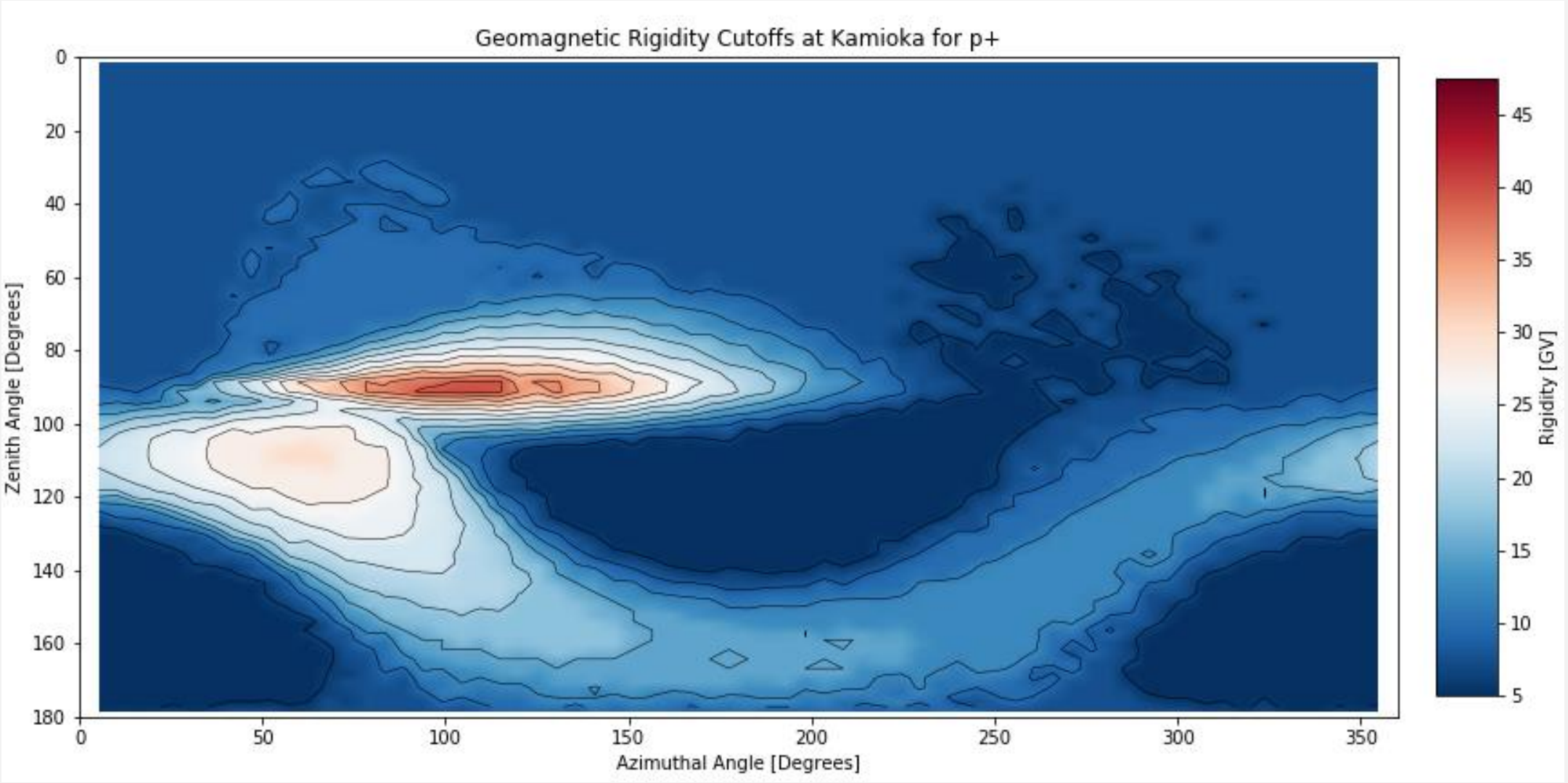
Function	Average Function Evaluation Duration [ns]
tan()	36.4514
$\frac{\sin()}{\cos()}$	25.3672

- Obtained from <cmath>
- Compiled with optimization flag –O2

https://www.valgrind.org/info/tools.html

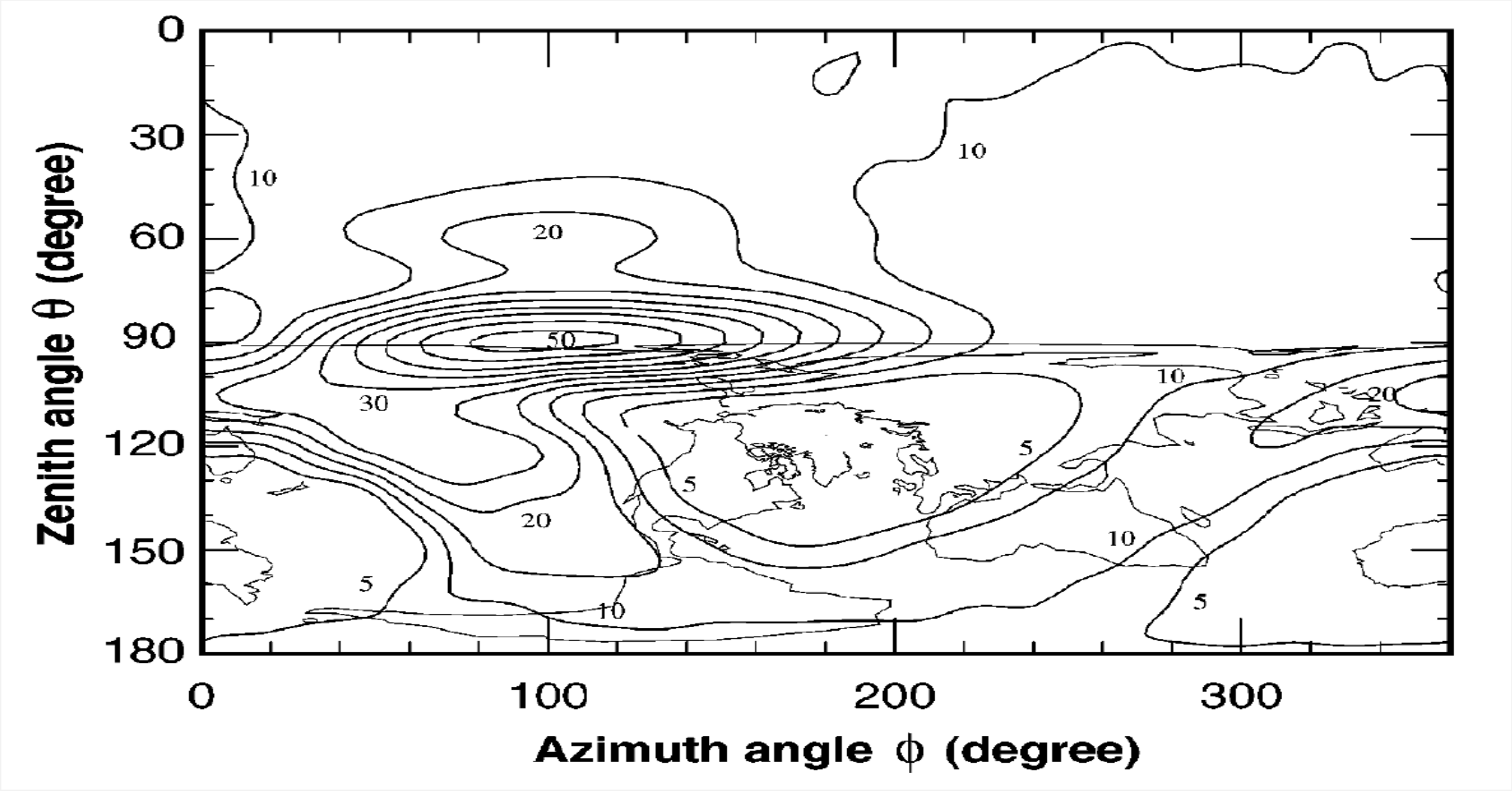
RESULTS

Geomagnetic Rigidity Cutoffs at Kamioka, Japan



RESULTS

Geomagnetic Rigidity Cutoffs at Kamioka, Japan from Honda, 2002



Annu. Rev. Nucl. Part. Sci. 2002.52:153-199.

OUTLOOK

- Optimize the physics involved in the calculation, ex. Reduce number of calls for magnetic field evaluation
- GPU acceleration to parallelize trajectory computations
- Use muon tracing for 3D neutrino flux calculations
- Implement continuous energy losses to the differential equation

BACK-UP SLIDES

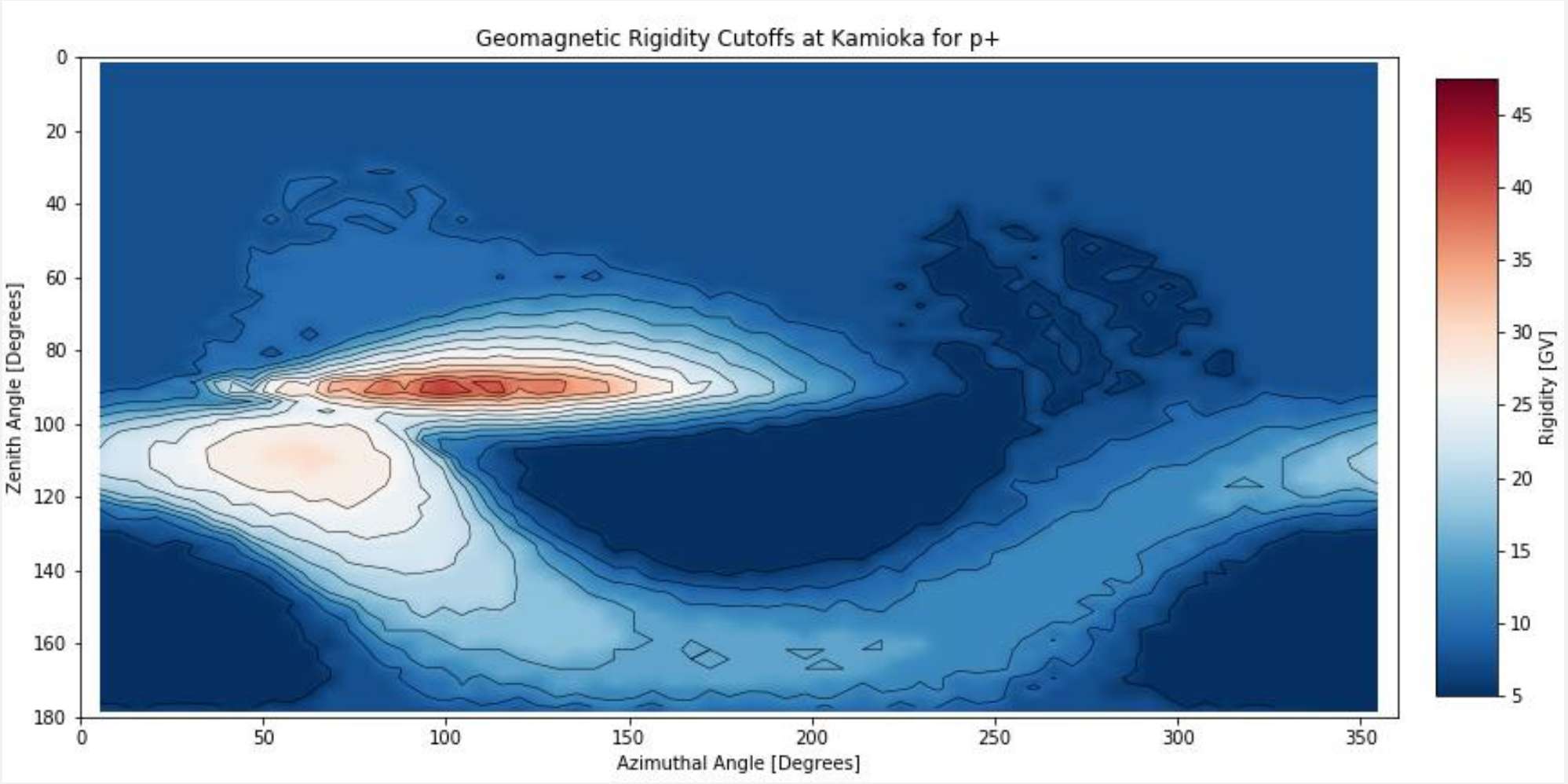
RESULTS: DIPOLE MODEL

$$V(r,\theta,\phi) = R_E\left(\frac{R_E}{r}\right)^2 g_1^0 \cos(\theta)$$

$$B_r(r,\theta,\phi) = -2g_1^0\left(\frac{R_E}{r}\right)^3 \cos(\theta);$$

$$B_\theta(r,\theta,\phi) = -g_1^0\left(\frac{R_E}{r}\right)^3 \sin(\theta);$$

$$B_\phi = 0$$



IGRF MODEL VS DIPOLE MODEL