Cockcroft-06-27

LARGE SIMULATION OF HIGH ORDER SHORT RANGE WAKEFIELDS

Adriana Bungau, The University of Manchester and Cockcroft Institute, UK and
Roger Barlow, The University of Manchester and Cockcroft Institute, UK

Abstract

We present a formalism for incorporating intra-bunch wake fields into particle-by-particle tracking codes, such as MERLIN and BDSIM. Higher order wake fields are incorporated
in a manner which is computationally efficient.
Standard formulae for geometric, resistive and dielectric wake fields are included for various apertures, particularly those relevant for ILC collimators.

Presented at the Tenth European Particle Accelerator Conference (EPAC06),
Edinburgh, UK, 26-30 June, 2006

LARGE SIMULATION OF HIGH ORDER SHORT RANGE WAKEFIELDS

Adriana Bungau, The University of Manchester and Cockcroft Institute, UK and
Roger Barlow, The University of Manchester and Cockcroft Institute, UK

Abstract

We present a formalism for incorporating intra-bunch wake fields into particle-by-particle tracking codes, such as MERLIN and BDSIM. Higher order wake fields are incorporated in a manner which is computationally efficient. Standard formulae for geometric, resistive and dielectric wake fields are included for various apertures, particularly those relevant for ILC collimators.

INTRODUCTION

There is an extensive literature on wake field effects and many programs for their calculation. Nevertheless the ILC collimation system raises new questions about wake fields which require an extension to the existing simulation tools. Existing literature concentrates on wake field effects in RF cavities, which have axial symmetry and in which particles are on or near the axis of symmetry so that only the lowest modes are important. Only long range wakefields are important as the cavities ring at particular frequencies. For the ILC collimators the short range wakes are the important ones. Particle bunches are distorted from their original Gaussian shape. The bunch passes close to the collimator so high order modes must be included. The collimator is not characterized by resonances and the system does not have axial symmetry. We find formulae for wake field effects in which higher modes are included, and show how these can be simulated in a program such as Merlin [1] which simulate individual wake effects with typically 10^5 acting particles on each other in an efficient way.

WAKE EFFECTS ON A SINGLE CHARGE

Consider the effect on a trailing particle at r, θ of a slice of N particles all ahead by the same distance s. We assume that all particles are relativistic ($v \approx c$, γ large) so that the direct effect of the charges on each other is suppressed by a power of γ and therefore ignored. So are effects in a uniform perfectly-conducting aperture: wake effects are ascribable to finite conductance or non-uniformity and are thus categorisable as resistive or geometric [2][3]. The effects of transverse velocity and acceleration during the passage of the particles through the aperture are ignored: r' and r are constant.
The total effect is given by simple summation. If we write $C_m = \Sigma r'^m cos(m\theta')$ and $S_m = \Sigma r'^m sin(m\theta')$ where the summation is over all particles in the slice, then the combined kick is

$$W_z = \Sigma W'_m(s) r^m [C_m cos(m\theta) + S_m sin(m\theta)] \qquad (1)$$

$$W_x = \Sigma m W_m(s) r^{m-1}\{C_m cos[(m-1)\theta] + S_m sin[(m-1)\theta]\} \qquad (2)$$

$$W_y = \Sigma m W_m(s) r^{m-1}\{S_m cos[(m-1)\theta] - C_m sin[(m-1)\theta\} \qquad (3)$$

These formulae for x and y, rather than the usual ones for 'transverse' wake, are correct even when the particles are spread out in azimuthal angle. The trigonometric terms are needed to describe a bunch which has not a compact (Gaussian) shape but has internal structure on the same scale as the beam pipe. The usual monopole and dipole formulae are reproduced for $m = 0, 1$. For a particle in slice i, a wakefield effect is received for all slices $j \geq i$. So the total effect for the x direction (for example) is:

$$\begin{aligned}\Sigma_j w_x &= \Sigma_j \Sigma_m m W_m(s_j) r^{m-1}\{C_{mj} cos[(m-1)\theta] \\ &\quad + S_{mj} sin[(m-1)\theta]\}\end{aligned} \qquad (4)$$

The summations can be re-ordered and the $\Sigma_j W_m(s_j) S_{mj}$ and $\Sigma_j W_m(s_j) C_{mj}$ terms can be calculated and stored:

$$\begin{aligned}\Sigma_j w_x &= \Sigma_m m r^{m-1}\{cos[(m-1)\theta]\Sigma_j W_m(s_j) C_{mj} \\ &\quad + sin[(m-1)\theta]\Sigma_j W_m(s_j) S_{mj}\}\end{aligned} \qquad (5)$$

IMPLEMENTATION IN MERLIN

This formalism is well suited to incorporation in a computer simulation code such as Merlin [1], as although the effect on a single particle requires summation over modes, within each mode the effect factors into: C_m and S_m which depend only on the leading slice, $W_m(s)$ which depends only on the aperture geometry and r and θ which depend only on the trailing particle.
At present the code contains two important and separate classes relevant to Wakefields: **WakePotentials** and **WakeFieldProcess**. Each component may contain a **WakePotentials** object. The class **WakePotentials** is defined in AcceleratorModel and contains the transverse and longitudinal wake functions **Wlong(z)** and **Wtrans(z)** and it also has a constructor. It is assigned to a particular accelerator component by a call to **AcceleratorComponent::SetWakePotentials(WakePotentials*)**. This is done by the user who usually invokes the constructor at the same time, and anything needed for the calculations (beam pipe radius or conductivity) is provided here in the argument list for the constructor: this is a point at which they are accessible. One might define (for example) **IrisWakePotentials:WakePotentials** and **PipeWakePotentials:WakePotentials** with different functional forms for **Wtrans** and **Wlong**.

Class **WakeFieldProcess** (defined in BeamDynamics/ParticleTracking) is derived from class **ParticleBunchProcess**. To run Merlin with wakefields, the user creates an instance of **WakeFieldProcess** and adds it to the list of processes of the tracker. The tracker is invoked for a particular beamline and bunch, and through the beamline, component by component, it tracks the bunch, particle by particle. This ordering of the loops makes wakefield implementation possible, as the configuration of all the particles is known as the bunch passes through each component. In tracking the bunch through a component, the program seeks to apply all ParticleBunchProcesses in its list, including WakeFieldProcess if it has been added. It invokes InitialiseProcess and splits the bunch into slices. **WakeFieldProcess** contains a pointer to a **WakePotentials** object. As the bunch is tracked through a component Merlin sets this pointer to the WakePotentials of the component in **WakeFieldProcess::SetCurrentComponent (AcceleratorComponent)**. If the component has no **WakePotentials** then various flags are set which ensure that nothing further happens. Merlin invokes the standard **DoProcess** which calls **ApplyWakeField**. For each slice, this sums the wake contributions from earlier slices (the Bunch Wake potentials) by calling **CalculateWakeT**, and **CalculateWakeL** using the **Wlong** and **Wtrans** functions of the **WakePotentials** and the charge distribution of the slice. These are stored in a table so need only be done once. It then applies the correct kick, particle by particle. This includes a numerical approximation using the gradient to allow for the longitudinal position of the particle within its slice.

The existing standard implementation only includes the monopole (longitudinal) and dipole (transverse) wakes. To include the higher order wakes important for collimators we have to add the ability to sum over modes. As MERLIN is an object-oriented program, such changes are essentially achievable by defining new derived classes containing the extra features desired. This ensures backward compatibility. We define new classes: **SpoilerWakePotentials** which inherits from **WakePotentials** and **SpoilerWakeFieldProcess** which inherits from **WakeFieldProcess**, detailed below. Each class contains a data member **int nmodes** which is the order to which the calculation is to be carried out. For **SpoilerWakePotentials** this is set when constructed, or by the user. For **SpoilerWakeFieldProcess** it is obtained from the relevant **SpoilerWakePotentials** as part of **SetCurrentComponent**. **SpoilerWakePotentials** is pure virtual. It has pure virtual functions **Wtrans(s,m)** and **Wlong(s,m)** which will be overridden in particular child classes such as **TaperedCollimator** and **StepCollimator**. **SpoilerWakeFieldProcess** is complete and does not require any subsequent filling in detail. All the flexibility is in **SpoilerWakePotentials**. It contains a new versions of **ApplyWakeField** with an extra loop over modes, and of **CalculateWakeT** and **CalculateWakeL.** Calculation of the moments S_m and C_m for each slice is done through new **CalculateSm** and **CalculateCm** routines..

MERLIN SIMULATIONS

The wake function for a steeply tapered collimator moving from aperture b to aperture a is given by the following expression [3]:

$$W_m(z) = 2\left(\frac{1}{a^{2m}} - \frac{1}{b^{2m}}\right) e^{\frac{-mz}{a}}\theta(z) \tag{6}$$

where $\theta(z)$ is a unit step function. This equation was implemented in the MERLIN code as follows:

```
class TaperedCollimatorPotentials:
          public SpoilerWakePotentials
{public:
 double a, b;
 double* coeff;
 TaperedCollimatorPotentials(int m, double
 rada, double radb): SpoilerWakePotentials
 (m, 0., 0.)
  {    a = rada;
       b = radb;
       coeff = new double [m];
 for (int i=0; i<m; i++) {
 coeff[i]=2*(1./pow(a,2*i)-1./pow(b,2*i));} }
 ~TaperedCollimatorPotentials(){delete[]coeff;}
 double Wlong (double z, int m) const {
 return z>0 ? -(m/a)*coeff[m]/exp(m*z/a):0;};
 double Wtrans (double z, int m) const {
 return z>0 ? coeff[m]/exp(m*z/a):0;}; };
```

SLAC beam tests were simulated with the new additions to the Merlin code. A Gaussian beam having an energy of 1.19 GeV and 2*10^{10} electrons was sent through a spoiler having a gap half-width of 1.9 mm. The beam emittance was set to ϵ_x=0.36 mm and ϵ_y=0.16 mm. The lattice functions were β_x=3m and β_y=10m, and the bunch length was set to σ_z=0.65 mm. Simulations were performed taking into account just the first mode for the start. Figure 1 shows the deflection in angle of the particles that emerge from the collimator. The y kick varies with the position along the bunch - the tail is more affected than the head - which will lead to non-Gaussian bunch shapes. When this effect becomes important, a particle by particle tracking code is essential. The wake effect at 0.5 mm offset for one mode is small and adding $m = 2, 3$ etc. does not change it much. However, for a large displacement of 1.5 mm, the bunch tail gets a bigger kick (figure 2) even when one mode is considered. Therefore, when the bunch offset is increasing higher order modes must be included in the simulations. An analysis with higher order modes was performed for 1.5 mm beam offset and figure 3 shows the results when three modes are considered. One can see that the tail gets a much bigger kick when three modes are included than in the case when only the first was considered. It is worth mentioning that to run a normal sample of 10^5 particles in a bunch, it normally takes a couple of minutes.

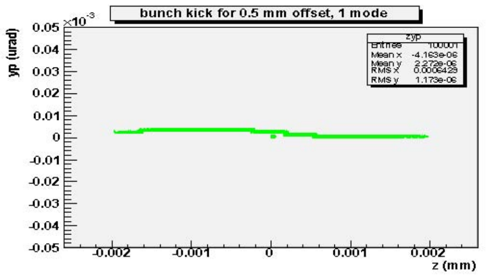

Figure 1: Bunch kick for 0.5 mm offset with one mode.

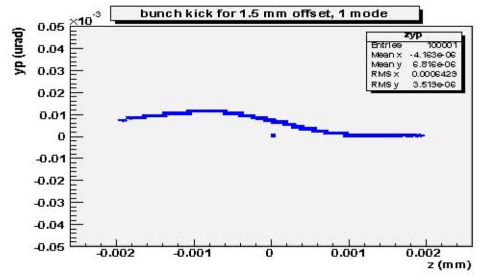

Figure 2: Bunch kick for 1.5 mm offset with one mode.

ADAPTATION TO NUMERIC SOLUTIONS

Codes such as Mafia and GDFIdL take a given charge and current distribution and compute the EM field by solving Maxwells equations. The wake potential can then be found by integrating along the z direction at a particular s after the original charge. From the form of the potential, we can do this for an arbitrary r and r' and several different values of θ, and then Fourier decompose the θ distribution to get the Fourier coefficients. Thus one can obtain a table for the the W_m at various different values of s, and write **Wlong(s,m)** and **Wtrans(s,m)** using interpolation for intermediate s values.

ACKNOWLEDGEMENTS

Thanks to Andy Wolski for setting us right on Merlin, and German Kourevlev and Adam Mercer for help at the start. This work is supported by the Commission of the European Communities under the 6^{th} Framework Programme "Structuring the European Research Area", contract number RIDS-011899.

REFERENCES

[1] MERLIN - A C++ Class Library for performing Charged Particle Accelerator Simulations, http://www.desy.de/∼merlin/

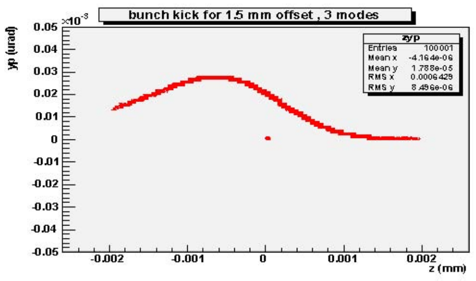

Figure 3: Bunch kick for 1.5 mm offset with three modes.

[2] G.Y.Stupakov, Wake and Impedance, arXIv:physics/0011011, 2000

[3] B.W.Zotter and S.A.Kheifets, Impedances and Wakes in High-Energy Particle Accelerators, World Scientific (1998)