Appendix A
Changes to Previous Publications

This chapter is ment for readers who followed previous publications or even based work of their own on these texts.

The following changes were made to the results given in previous publications in comparison to the presentation in this document:

European Convention The papers describing the theoretical development used the US convention as it is more close to standard Fourier Series. European convention (see (2.42) p. 26) is used throughout this text (original papers [1–7]).

Toroidal multipoles The complex representation of the basis function and the calculation of the real terms was changed. Here concise descriptions of the basis functions are given. The representation chosen now allows describing the basis functions of the potential and the field as complex functions. However these are not analytic and different functions have to be used for the "normal" and "skew" coefficients (original papers [7]).

Measuring toroidal multipoles The theory for measuring toroidal circular multipoles with a rotating coil probe was outlined in [7]; there the area normal of the coil probe was not taken properly into account. This is corrected in this treatise.

P. Schnizer, *Advanced Multipoles for Accelerator Magnets*, Springer Tracts in Modern Physics 277, DOI 10.1007/978-3-319-65666-3

Appendix B
Mathematica Scripts

B.1 Coil Probe Within a Torus

The scripts, listed below, were used to calculate the field which a rotating coil probe measures inside a torus. The field is described with local toroidal cylindrical multipoles as given in (4.122), p. 63.

The calculations of the integrals of the measurement coil probe and the matrices to interpret these measurements were presented in Sect. 8.2, p. 116. The following script was used to evaluate the integral and the next one to check that the matrix prediction was correct. The scripts were split as the integral evaluation required approximately 3 hours on contemporary hardware with up to 16 processes evaluating the integrals in parallel.

evaluate_integrals_parallel.m

```
(* ::Package:: *)

(* ============================================================ *)
(* The local toroidal basis functions *)
TmComplexCommonTerm[zs_, eps_, n_] :=
    (zs)^(n - 1) * (1 - 1 / 2 * eps * Re[zs]);

TmComplexTerm[zs_, eps_, n_] :=
Module[{common, notAnalytic, result},
    common = TmComplexCommonTerm[zs, eps, n];
    notAnalytic = - eps  / (2 * n) *
        {Im[zs^(n - 1) * zs] *I, Re[zs^(n - 1) * zs]};
    result = common + notAnalytic;
    result
];

(* ============================================================ *)
(* helper functions *)
variabelsAndCoil[expr_] :=
    Block[{subs, result},
    subs = (r/R * Exp[I * theta] + dx/R + dy/R * I +
```


P. Schnizer, *Advanced Multipoles for Accelerator Magnets*, Springer Tracts in Modern Physics 277, DOI 10.1007/978-3-319-65666-3

```
            eps * z^2 / R^2);
    result = ReplaceAll[expr,  zs -> subs]* Exp[I * theta];
    result
]

integrateOverCoil[expr_] :=
    Block[{realVars},
    realsVars={R, r, r1, r2, theta, z, L, eps, dy, dx};
    Integrate[expr, {z, 0, L},{r, r1, r2},
        Assumptions->Element[realVars, Reals] &&
        R > 0 && eps >0 && r2 > r1]
]

buildTerms[expr_, start_Integer, end_Integer, normalOrSkew_] :=
 Block[{termNormal, fluxNormal, flx},
    termNormal = expr;
    fluxNormal = variabelsAndCoil[termNormal];
    flx = Table[fluxNormal, {n, start, end}];
    flx = ComplexExpand[Re[flx * normalOrSkew]];
    flx = ReplaceAll[flx, onlyLinearInEps];
    flx
    ]

(* The integrals *)
(* ================================================== *)
minterm = 1
maxterm = 21
onlyLinearInEps = Table[eps^n -> 0,
    {n, 2, 2 * Max[minterm, maxterm]}];
savename = ("pierres_integral_"
    <> ToString[minterm] <> "_" <> ToString[maxterm])
Print[savename];

basisterms = TmComplexTerm[zs, eps, n]

(* The normal components *)
flxn = buildTerms[basisterms[[1]], minterm, maxterm, 1];

(* The skew components *)
flxs = buildTerms[basisterms[[2]], minterm, maxterm, I];

(* Revert the list ... faster parallel execution .... *)
flxn = flxn[[-1;;1;;-1]];
flxs = flxs[[-1;;1;;-1]];
(* Prepare to evaluate all at once *)
allflx = Apply[Join, Transpose[{flxn, flxs}]];
dims = Dimensions[allflx];
If[Length[dims] == 1, Null,
    Print["List of integrands not one dimensional!", dims];
    Exit[]];
ninteg = Length[allflx];
Print["Starting integral evaluation ", ninteg, " integrands"];
{usedTime, integralall} = AbsoluteTiming[
```

```
    ParallelMap[integrateOverCoil, allflx]
];
Print["Finished: required ", N[usedTime,1], "seconds"];

(* separate the result again ... *)
(*   normal *)
integraln = integralall[[1;; ;; 2]]
(*   skew *)
integrals = integralall[[2;; ;; 2]]
integraln = integraln[[-1;;1;;-1]];
integrals = integrals[[-1;;1;;-1]];

(* and store the result for later analysis *)
Save[savename, {integraln, integrals}];
```

check_integrals_all1.m

```
(* ::Package:: *)

(*
 * Check if the guessed representation of the integrals matches the
 * integrals achieved with mathematica
 *)
(* ========================================================== *)
On[Assert];
(*============================================================*)
(*
 * The matrix U has to be evaluated to more one colum than any
 * other matrix here. This function adds a zero colume to a given
 * matrix
*)
addColumnToMat[mat_] :=
Block[{dims, nrows, tcol, result},
    dims = Dimensions[mat];
    Assert[Length[dims] == 2];
    nrows = dims[[1]];
    tcol = Array[Function[{trow}, 0], nrows];
    result = Append[Transpose[mat], tcol];
    result = Transpose[result];
    result
]
(* for convienience ... *)
identityMatrix[nterms_Integer] :=
    addColumnToMat[IdentityMatrix[nterms]]

checkZeroInt[val_] := (MatchQ[val, _Integer] && val  == 0)

checkZeroVector[mat_] :=
Module[
    {result, nrows, trow, dims,tmp },
    dims = Dimensions[mat];
    nrows = dims[[1]];
    (* check every value if it is an integer *)
    tmp= Table[checkZeroInt[Part[mat, trow]],{trow, nrows}];
```

```
    (* now lets see if all were True! *)
    result = Apply[And,Map[Apply[And, #]&, tmp]];
    result
]

createFeedDownMatrix[dzs_, n_Integer, default_:0] := Module[
    {feeddowntable, result},
    feeddowntable = createLowerMat[
        Function[{trow, tcol},
            Binomial[trow-1,tcol-1] *(dzs)^(trow - tcol)],
        n, n, Null, default
    ];
    result = feeddowntable;
    result
]

createLowerMat[f_, nrows_Integer, ncols_Integer, args_, default_:0] :=
Module[
    {trow, tcol, result},
    result = Array[
        Function[{trow, tcol},
            If[trow >= tcol, f[trow, tcol, args] , default]
          ],
        {nrows, ncols}
    ];
    result
]

(*============================================================ *)
Get["pierres_integral_1_21"];
integralnt = integraln;
integralst = integrals;
Dimensions[integralnt]
(* replacement lists to get from coil sensitity to R2 and R1 etc *)
invcoilSensitivities=Table[
    sk[n]->(r2^n- r1^n)* L/R^(n-1)/n,{n, 1, Length[integralnt]*2}];

createFeedDownMatrixPaper[dzs_, n_Integer, default_:0] :=
    createFeedDownMatrix[dzs, n, default] - IdentityMatrix[n];

(* The different sub matrices *)
(* The most difficult term  *)
ConstructR2MatCheck[nterms_] :=
Module[
{trow, tcol, nrows, ncols, lowermat, feeddownmat, feeddownscale, dims,
    upperdiagonal, corrections, result, coeffsreal, coeffsupper,
    coeffsimag, coeffsscale},
nrows = nterms;
ncols = nterms;
dims = {nrows, ncols};
(* The part similar scaling with dz / R *)
feeddownmat =  createFeedDownMatrixPaper[dx/R + I * dy/R, nterms] +
 IdentityMatrix[nterms];
```

```
feeddownscale = Array[Function[{trow, tcol}, 1/trow], {nrows, ncols}];
result = feeddownscale * feeddownmat / 4 ;

(* now comes this peculiar part in dx and dy .... *)
(* the coefficients *)
coeffsreal = createLowerMat[
        Function[{trow, tcol}, trow * tcol / (trow - tcol + 1)],
    nrows, ncols, Null];

coeffsreal = (coeffsreal +
    Array[Function[{trow, tcol}, KroneckerDelta[tcol, 1 ]], dims]);

coeffsscale = createLowerMat[
        Function[{trow, tcol},(2 - tcol + 2 * trow) / (tcol) ],
    nrows, ncols, Null];
coeffsscale += Array[Function[{trow, tcol},
    - trow * KroneckerDelta[tcol, 1]],dims];

corrections = coeffsreal ( dy / R -coeffsscale* dx /R * I) ;
result = result * corrections ;
result = Expand[result];
addColumnToMat[result]
]
```

```
Ldr [nterms_Integer] := Expand[
    addColumnToMat[
    createFeedDownMatrixPaper[(dx/R + I * dy/R), nterms, 0]
    ]
]

U[nterms_Integer] :=
 Array[Function[{trow, tcol},
    KroneckerDelta[trow + 1, tcol] * (trow +1) / ((trow) * 4 )],
    {nterms, nterms+1}
]

createFeedDownDerivMatrix[dzs_, n_Integer, default_:0] :=
    Block[
    {fdDzs, dfdDzs, tdzs},
        fdDzs = createFeedDownMatrixPaper[tdzs, n, default];
        dfdDzs = D[fdDzs, tdzs];
        dfdDzs = ReplaceAll[dfdDzs, tdzs -> dzs];
        addColumnToMat[dfdDzs]
]

LL[nterms_Integer] :=
     Block[
    {result, dfdDzs},
    dfdDzs = createFeedDownDerivMatrix[dx/R + dy/R*I, nterms, 0];
    result = (L^2 / (R^2 * 3)) (dfdDzs);
    result = Expand[result];
    result
]
```

```
Lsk[nterms_Integer] :=
Block[{result, scale, dfdDzs},
    scale = createLowerMat[
        Function[{trow, tcol, unused},
                    1/4 * 1/(tcol+1) *  sk[tcol+2]/sk[tcol]],
     nterms, nterms+1, Null];
    dfdDzs = createFeedDownDerivMatrix[dx/R + dy/R*I, nterms, 0];
    result = scale *  dfdDzs;
    result = Expand[result];
    result
]

LR2[nterms_Integer] := ConstructR2MatCheck[nterms]

LR20[nterms_Integer] := addColumnToMat[
  Array[
    Function[{trow, tcol},
      KroneckerDelta[tcol, 1] * 1 /( 2 * trow )* ((dx + I * dy)/R)^(trow)],
    {nterms, nterms}]
]

toroidalNormalToCoilNormal[ndim_] :=
(identityMatrix[ndim]+Re[Ldr[ndim]]
  +eps(-U[ndim]+Re[LL[ndim]]-Re[Lsk[ndim]]+Im[LR2[ndim]]+Re[LR20[ndim]]))

toroidalNormalToCoilSkew[ndim_] :=
(                    +Im[Ldr[ndim]]
   +eps(         +Im[LL[ndim]]-Im[Lsk[ndim]]-Re[LR2[ndim]]))

toroidalSkewToCoilNormal[ndim_] :=
(                     -Im[Ldr[ndim]]
  +eps(          -Im[LL[ndim]]+Im[Lsk[ndim]]+Re[LR2[ndim]]-Im[LR20[ndim]]))

toroidalSkewToCoilSkew[ndim_] :=
(identityMatrix[ndim]+Re[Ldr[ndim]]
  + eps(-U[ndim]+Re[LL[ndim]]-Re[Lsk[ndim]]+Im[LR2[ndim]]));

constructIntegrals[mat_, term_] :=
Block[{tmp, nterms, dims, termtable, result},
    dims = Dimensions[mat];
    Assert[Length[dims] == 2];
    nterms = dims[[2]];
    termtable = Table[term * sk[n], {n, 1, nterms}];
    tmp = mat. termtable;
    tmp = ReplaceAll[tmp, invcoilSensitivities];
    result = Expand[tmp];
    result
    ]

calculateNDim[mat_]:= Max[Dimensions[mat]]
selectOnlyFirstRows[mat_, dims_] :=
Block[{result, tdims, nrows, ncols},
```

```
    Assert[Length[dims] == 2];
    tdims = Dimensions[mat];
    Assert[Length[tdims] == 2];
    nrows = dims[[1]];
    ncols = dims[[2]];
    Assert[nrows <= tdims[[1]]];
    Assert[ncols <= tdims[[2]]];
    result = Part[mat, ;; nrows, ;; ncols];
    result;
]

ndim = calculateNDim[integralnt];
Print["Building normal component terms up to term ", ndim];
refdim = Dimensions[integralnt];
convnn = constructIntegrals[
    ComplexExpand[toroidalNormalToCoilNormal[ndim]], Cos[n * theta]];
(*
*   the terms in Sin have to be taken times -1 as
* Phi = ComplexExpand[Re[(Bn + I * An) * Exp[I * n * theta]]]
* Phi = Bn Cos[n theta]-An Sin[n theta]
*)
convns = constructIntegrals[
    ComplexExpand[toroidalNormalToCoilSkew[ndim]],      Sin[n * theta]];
convns = convns * -1;
Print[Dimensions[convnn]];
Print["Comparing normal component terms to integrals"];
result = Expand[TrigReduce[integralnt]] - convnn - convns;
Print["   Simplifying Diff"];
result = Simplify[result];
checkn = checkZeroVector[result];
If[checkn == True,
    Print["Integral for normal components fully represented (expected!)"],
    Print["Representing the integral for the normal components left over:",
    result, checkn]
];
Print["Building skew component terms"];
ndim = calculateNDim[integralnt];
convsn = constructIntegrals[
    ComplexExpand[toroidalSkewToCoilNormal[ndim]], Cos[n * theta]];
convss = constructIntegrals[
    ComplexExpand[toroidalSkewToCoilSkew[ndim]], Sin[n * theta]];
(*
 * again ... the coefficients have to be multiplied -1 as the normal
 * coil probesees the field - Sin[n * theta]
 *)
convss = convss * -1;
Print[Dimensions[convsn]];
Print["Comparing skew component terms to integrals"];
result = Expand[TrigReduce[integralst]] - convsn - convss;
Print["   Simplifying Diff"];
result = Simplify[result];
checks = checkZeroVector[result];
If[checks == True,
```

```
Ldr [nterms_Integer] := Expand[
    addColumnToMat[
    createFeedDownMatrixPaper[(dx/R + I * dy/R), nterms, 0]
    ]
]

U[nterms_Integer] :=
 Array[Function[{trow, tcol},
    KroneckerDelta[trow + 1, tcol] * (trow +1) / ((trow) * 4 )],
    {nterms, nterms+1}
]

createFeedDownDerivMatrix[dzs_, n_Integer, default_:0] :=
    Block[
    {fdDzs, dfdDzs, tdzs},
        fdDzs = createFeedDownMatrixPaper[tdzs, n, default];
        dfdDzs = D[fdDzs, tdzs];
        dfdDzs = ReplaceAll[dfdDzs, tdzs -> dzs];
        addColumnToMat[dfdDzs]
]

LL[nterms_Integer] :=
     Block[
    {result, dfdDzs},
    dfdDzs = createFeedDownDerivMatrix[dx/R + dy/R*I, nterms, 0];
    result = (L^2 / (R^2 * 3)) (dfdDzs);
    result = Expand[result];
    result
]

Lsk[nterms_Integer] :=
Block[{result, scale, dfdDzs},
    scale = createLowerMat[
        Function[{trow, tcol, unused},
                    1/4 * 1/(tcol+1) *  sk[tcol+2]/sk[tcol]]],
     nterms, nterms+1, Null];
    dfdDzs = createFeedDownDerivMatrix[dx/R + dy/R*I, nterms, 0];
    result = scale *  dfdDzs;
    result = Expand[result];
    result
]

LR2[nterms_Integer] := ConstructR2MatCheck[nterms]

LR20[nterms_Integer] := addColumnToMat[
  Array[
    Function[{trow, tcol},
      KroneckerDelta[tcol, 1] * 1 /( 2 * trow )* ((dx + I * dy)/R)^(trow)],
    {nterms, nterms}]
]

toroidalNormalToCoilNormal[ndim_] :=
(identityMatrix[ndim]+Re[Ldr[ndim]]
```

```
  +eps(-U[ndim]+Re[LL[ndim]]-Re[Lsk[ndim]]+Im[LR2[ndim]]+Re[LR20[ndim]]))

toroidalNormalToCoilSkew[ndim_] :=
(                        +Im[Ldr[ndim]]
   +eps(          +Im[LL[ndim]]-Im[Lsk[ndim]]-Re[LR2[ndim]]))

toroidalSkewToCoilNormal[ndim_] :=
(                          -Im[Ldr[ndim]]
  +eps(          -Im[LL[ndim]]+Im[Lsk[ndim]]+Re[LR2[ndim]]-Im[LR20[ndim]]))

toroidalSkewToCoilSkew[ndim_] :=
(identityMatrix[ndim]+Re[Ldr[ndim]]
  + eps(-U[ndim]+Re[LL[ndim]]-Re[Lsk[ndim]]+Im[LR2[ndim]]));

constructIntegrals[mat_, term_] :=
Block[{tmp, nterms, dims, termtable, result},
    dims = Dimensions[mat];
    Assert[Length[dims] == 2];
    nterms = dims[[2]];
    termtable = Table[term * sk[n], {n, 1, nterms}];
    tmp = mat. termtable;
    tmp = ReplaceAll[tmp, invcoilSensitivities];
    result = Expand[tmp];
    result
    ]

calculateNDim[mat_]:= Max[Dimensions[mat]]
selectOnlyFirstRows[mat_, dims_] :=
Block[{result, tdims, nrows, ncols},
    Assert[Length[dims] == 2];
    tdims = Dimensions[mat];
    Assert[Length[tdims] == 2];
    nrows = dims[[1]];
    ncols = dims[[2]];
    Assert[nrows <= tdims[[1]]];
    Assert[ncols <= tdims[[2]]];
    result = Part[mat, ;; nrows, ;; ncols];
    result;
]

ndim = calculateNDim[integralnt];
Print["Building normal component terms up to term ", ndim];
refdim = Dimensions[integralnt];
convnn = constructIntegrals[
    ComplexExpand[toroidalNormalToCoilNormal[ndim]], Cos[n * theta]];
(*
*   the terms in Sin have to be taken times -1 as
* Phi = ComplexExpand[Re[(Bn + I * An) * Exp[I * n * theta]]]
* Phi = Bn Cos[n theta]-An Sin[n theta]
*)
convns = constructIntegrals[
    ComplexExpand[toroidalNormalToCoilSkew[ndim]],      Sin[n * theta]];
convns = convns * -1;
```

```
Print[Dimensions[convnn]];
Print["Comparing normal component terms to integrals"];
result = Expand[TrigReduce[integralnt]] - convnn - convns;
Print["   Simplifying Diff"];
result = Simplify[result];
checkn = checkZeroVector[result];
If[checkn == True,
    Print["Integral for normal components fully represented (expected!)"],
    Print["Representing the integral for the normal components left over:",
    result, checkn]
];
Print["Building skew component terms"];
ndim = calculateNDim[integralnt];
convsn = constructIntegrals[
    ComplexExpand[toroidalSkewToCoilNormal[ndim]], Cos[n * theta]];
convss = constructIntegrals[
    ComplexExpand[toroidalSkewToCoilSkew[ndim]], Sin[n * theta]];
(*
 * again ... the coefficients have to be multiplied -1 as the normal
 * coil probesees the field - Sin[n * theta]
 *)
convss = convss * -1;
Print[Dimensions[convsn]];
Print["Comparing skew component terms to integrals"];
result = Expand[TrigReduce[integralst]] - convsn - convss;
Print["   Simplifying Diff"];
result = Simplify[result];
checks = checkZeroVector[result];
If[checks == True,
```

```
    Print["Integral for skew components fully represented (expected!)"],
    Print["Representing the integral for the normal components left over:",
     result, checks]
];

Print["Reached end of Programm"];
```

Appendix C
Approximate Inversion of a Perturbed Matrix

A matrix C is considered, which can be split into two matrices A and B by

$$C = A + eB\,. \tag{C.1}$$

It is assumed that the inverse of the matrix A is known and that

$$|eB| \ll |A| \tag{C.2}$$

so that eB can be seen as perturbation of the matrix A. In this case the inverse C^{-1} is given by

$$C^{-1} = (A + eB)^{-1} = A^{-1} - eA^{-1}BA^{-1} + \mathcal{O}\left(e^2\right)\,. \tag{C.3}$$

This can be proven by

$$\begin{aligned}
(A + eB)(A + eB)^{-1} &= (A + eB)\left(A^{-1} - eA^{-1}BA^{-1} + \mathcal{O}\left(e^2\right)\right) && \text{(C.4)}\\
&= AA^{-1} + e(BA^{-1} - AA^{-1}BA^{-1}) + \mathcal{O}\left(e^2\right) && \text{(C.5)}\\
&= E + e(BA^{-1} - EBA^{-1}) + \mathcal{O}\left(e^2\right) && \text{(C.6)}\\
&= E + \mathcal{O}\left(e^2\right)\,. && \text{(C.7)}
\end{aligned}$$

Similarly

$$\begin{aligned}
(A + eB)^{-1}(A + eB) &= \left(A^{-1} - eA^{-1}BA^{-1} + \mathcal{O}\left(e^2\right)\right)(A + eB) && \text{(C.8)}\\
&= A^{-1}A + e\left(-A^{-1}BA^{-1}A + A^{-1}B\right) + \mathcal{O}\left(e^2\right) && \text{(C.9)}\\
&= E + e\left(-A^{-1}BE + A^{-1}B\right) + \mathcal{O}\left(e^2\right) && \text{(C.10)}\\
&= E + \mathcal{O}\left(e^2\right)\,. && \text{(C.11)}
\end{aligned}$$

P. Schnizer, *Advanced Multipoles for Accelerator Magnets*, Springer Tracts in Modern Physics 277, DOI 10.1007/978-3-319-65666-3

References

1. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Field representation for elliptic apertures. Technical report, Gesellschaft für Schwerionenforschung mbH, Planckstraße 1, D-64291 Darmstadt, February 2007
2. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Magnetic field analysis for superferric accelerator magnets using elliptic multipoles and its advantages. IEEE T. Appl. Supercon. **18**(2), 1605–1608 (2008)
3. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Theoretical field analysis for superferric accelerator magnets using plane elliptic or toroidal multipoles and its advantages, in *The 11th European Particle Accelerator Conference*, June 2008, pp. 1773–1775
4. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Field representation for elliptic apertures. Technical report, Gesellschaft für Schwerionenforschung mbH, Planckstraße 1, D-64291 Darmstadt, January 2008
5. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Theory and application of plane elliptic multipoles for static magnetic fields. Nucl. Instrum. Methods Phys. Res. Sect. A: Accel. Spectrometers, Detectors Assoc. Equip. **607**(3), 505–516 (2009)
6. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Toroidal circular and elliptic multipole expansions within the gap of curved accelerator magnets, in *14th International IGTE Symposium, Graz, Institut für Grundlagen und Theorie der Elektrotechnik* (Technische Universität Graz, Austria, September 2010)
7. P. Schnizer, B. Schnizer, P. Akishin, E. Fischer, Plane elliptic or toroidal multipole expansions for static fields. Applications within the gap of straight and curved accelerator magnets. Int. J. Comput. Math. Electr. Eng. (COMPEL) **28**(4) (2009)