

RB-SFA: High Harmonic Generation in the Strong Field Approximation via *Mathematica*

© Emilio Pisanty 2014-2017. Licensed under GPL and CC-BY-SA.

■ Introduction

Readme

RB-SFA is a compact and flexible Mathematica package for calculating High Harmonic Generation emission within the Strong Field Approximation. It combines Mathematica's analytical integration capabilities with its numerical calculation capacities to offer a fast and user-friendly plug-and-play solver for calculating HHG spectra and other properties. In addition, it can calculate first-order nondipole corrections to the SFA results to evaluate the effect of the driving laser's magnetic field on harmonic spectra. There is also an experimental section for calculating spectra using quantum-orbit methods.

The name RB-SFA comes from its first application (as Rotating Bicircular High Harmonic Generation in the Strong field Approximation) but the code is general so RB-SFA just stands for itself now. The publications by the author that use this code are:

- Strong-field approximation in a rotating frame: high-order harmonic emission from p states in bicircular fields. E. Pisanty and Á. Jiménez-Galán. *Phys. Rev. A* **96** 063401 (2017), arXiv:1709.00397.
- Electron dynamics in complex time and complex space. E. Pisanty Alatorre. PhD Thesis, Imperial College London, 2016.
- High harmonic interferometry of the Lorentz force in strong mid-infrared laser fields. E. Pisanty, D.D. Hickstein, et al. (2016), arXiv:1606.01931.
- Spin conservation in high-order-harmonic generation using bicircular fields. E. Pisanty, S. Sukiasyan and M. Ivanov. *Phys. Rev. A* **90**, 043829 (2014), arXiv:1404.6242.

This code is dual-licensed under the GPL and CC-BY-SA licenses; you are free to use, modify, and redistribute it, but you must abide by the terms in either of those licenses.

In addition to that *legal* obligation, if you use this code in calculations for an academic publication, you have an *academic* obligation to cite it correctly. For that purpose, please cite the PhD thesis above, or use a direct citation to the code such as

E. Pisanty. RB-SFA: High Harmonic Generation in the Strong Field Approximation via *Mathematica*. <https://github.com/episanty/RB-SFA> (2016).

If you wish to include a DOI in your citation, please use one of the numbered-version releases.

This software consists of the Mathematica notebook `RB-SFA.nb`, which contains the code and its documentation, a corresponding auto-generated package file `RB-SFA.m`, which provides the package functions to other notebooks, a `Usage and Examples.nb` notebook which explains how to install and use the code, and documents the calculations used in the original publication, and a (draft) `Quantum Orbits Usage.nb` notebook documenting the use of the quantum-orbit functionality. PDF printouts of all notebooks are also provided.

```
In[ ]:= (*
This is the RB-SFA package for calculating high-
order harmonic generation within the Strong Field Approximation. For the
notebook that generated this package file and additional documentaion,
see https://github.com/episanty/RB-SFA.
*)
```

■ Implementation

Supporting functions

Initialization

```
In[ ]:= BeginPackage["RBSFA`"];
```

Version number

The command `RBSFAversion` prints the version of the RB-SFA package currently loaded and its timestamp

```
In[ ]:= $RBSFAversion::usage = "$RBSFAversion prints the
current version of the RB-SFA package in use and its timestamp.";
$RBSFAtimestamp::usage = "$RBSFAtimestamp prints the timestamp of
the current version of the RB-SFA package.";
Begin["`Private`"];
$RBSFAversion := "RB-SFA v2.1.3, " <> $RBSFAtimestamp;
End[];
```

Old syntax (in functional form `RBSFAversion[]`), deprecated

```
In[ ]:= RBSFAversion::usage = "RBSFAversion[] has been deprecated in favour of $RBSFAversion.";
RBSFAversion::dprc = "RBSFAversion[] has been deprecated in favour of $RBSFAversion.";
Begin["`Private`"];
RBSFAversion[] := (Message[RBSFAversion::dprc]; $RBSFAversion);
End[];
```

The timestamp is updated every time the notebook is saved via an appropriate notebook option, which is set by the code below.

```
SetOptions[
  EvaluationNotebook[],
  NotebookEventActions → {"MenuCommand", "Save"} → (
    NotebookWrite[
      Cells[CellTags → "version-timestamp"][[1]],
      Cell[
        BoxData[
          RowBox[{"Begin[\"`Private`\"];\n$RBSFAtimestamp=\"\" <> DateString[] <> \"\";\nEnd[];"}]]
        , "Input", InitializationCell → True, CellTags → "version-timestamp"
      ], None, AutoScroll → False];
    NotebookSave[]
  ), PassEventsDown → True}
];
```

To reset this behaviour to normal, evaluate the cell below

```
SetOptions[EvaluationNotebook[],
  NotebookEventActions → {"MenuCommand", "Save"} → (NotebookSave[], PassEventsDown → True)]
```

Timestamp

```
Begin["`Private`"];
$RBSFAtimestamp = "Fri 21 Feb 2020 16:56:05";
End[];
```

Directory

In[]:=

```
$RBSFAdirectory::usage =
  "$RBSFAdirectory is the directory where the current RB-SFA package instance is located.";
```

```

In[ ]:= Begin["`Private`"];
With[{softLinkTestString = StringSplit[StringJoin[
    ReadList["! ls -la " <> StringReplace[$InputFileName, {" " → "\\ "}], String]], " -> "],
    If[Length[softLinkTestString] > 1, (*Testing in case $InputFileName
        is a soft link to the actual directory.*)
        $RBSFAdirectory = StringReplace[DirectoryName[softLinkTestString[[2]], {" " → "\\ "}],
        $RBSFAdirectory = StringReplace[DirectoryName[$InputFileName], {" " → "\\ "};
    ]];
End[];

```

Git commit hash and message

```

In[ ]:= $RBSFAcommit::usage = "$RBSFAcommit returns the git
    commit log at the location of the RB-SFA package if there is one.";
$RBSFAcommit::OS = "$RBSFAcommit has only been tested on Linux.";

```

```

In[ ]:= Begin["`Private`"];
$RBSFAcommit := (If[$OperatingSystem ≠ "Unix", Message[$RBSFAcommit::OS]];
    StringJoin[Riffle[ReadList["!cd " <> $RBSFAdirectory <> " && git log -1", String], {"\n"}]]);
End[];

```

Standard function (re)definitions

ConstantArray

This redefines ConstantArray to take the corner case of an empty dimensions list, which returns an error code (and an unevaluated ConstantArray) for Mathematica versions under 10.1.0 (cf. mma.se/q/133078).

```

In[ ]:= Quiet[Check[
    ConstantArray[0, {}];,
    Unprotect[ConstantArray];
    ConstantArray[Private`x_, {}] := Private`x;
    Protect[ConstantArray];
]];

```

Similarly, this needs to be put inside an initialization code for any parallelized subkernels that may get launched later (cf. mm.se/q/131856).

```
In[ ]:= Parallelize;
Parallel`Developer`$InitCode = Hold[
  Quiet[Check[
    ConstantArray[0, {}];,
    Unprotect[ConstantArray];
    ConstantArray[Private`x_, {}] := Private`x;
    Protect[ConstantArray];
  ]];
];
```

ReIm

This adds the definition of ReIm for those versions (<10.1) that don't have it.

```
In[ ]:= If[
  Context[ReIm] != "System`" && Attributes[ReIm] == {},
  ReIm::usage =
    "\!\(\*RowBox[{\\"ReIm\", \\"[\", StyleBox[\\\"z\\\", \\\"TI\\\"], \\"\\\"}]\) gives the list
    \!\(\*RowBox[{\\"{\\\", RowBox[{\\"RowBox[{\\"Re\\\", \\"[\", StyleBox[\\\"z\\\",
    \\\"TI\\\", \\"\\\"}], \",\\\", RowBox[{\\"Im\\\", \\"[\", StyleBox[\\\"z\\\", \\\"TI\\\",
    \\"\\\"}]]], \\"\\\"}]\) of the number \!\(\*StyleBox[\\\"z\\\", \\\"TI\\\"]\).";
  ReIm[Private`z_] := {Re[Private`z], Im[Private`z]};
  SetAttributes[ReIm, Listable];
  Protect[ReIm];
];
```

AssociationTranspose

ClearAll[AssociationTranspose]

```
In[ ]:= AssociationTranspose::usage = "AssociationTranspose[association]
  transposes the given two-level association of associations.";
AssociationTranspose::wrngshp = "Input `1` is the wrong shape; it must be
  an association all of whose Values are valid associations.";
Begin["`Private`"];
AssociationTranspose[association_?(
  And @@ (AssociationQ /@ Join[{{#}}, Values[{{#}}]) &
)] := GroupBy[
  Join @@ Thread /@ Normal /@ association,
  {First@*Last, First}
][[All, All, 1, 2, 2]];
AssociationTranspose[association_] := "Doesn't display; cf. mm.se/q/29321 for details" /;
Message[AssociationTranspose::wrngshp, association]
End[];
```

The above function is taken from <http://mathematica.stackexchange.com/a/86526> by <http://mathematica.stackexchange.com/users/121/mr-wizard>. *Mathematica* 10.1 and higher, it can be replaced by a `Query[Transpose]` construct as below, but *Mathematica* 10.0, despite having most of the Association code, is unable to transpose ragged associations using that construct.

```
(*
AssociationTranspose[association_]:=DeleteMissing[
  Query[Transpose][
    association
  ]
  ,2]
*)
```

KeyValueMap

This exists in *Mathematica* 10.1 and later, but it's nice to have it on version 10.0 so this is a back-port for versions that do not have it.

```

In[ ]:=
If[
  $VersionNumber < 10.1,
  KeyValueMap::usage =
    "\\(\\*RowBox[{\"KeyValueMap\", \"[\", RowBox[{StyleBox[\"f\", \"TI\"], \",\", \"\",
      RowBox[{\"<\", RowBox[{RowBox[{SubscriptBox[StyleBox[\"key\", \"TI\"], StyleBox[\"1\",
        \"TR\"]}], \"→\", SubscriptBox[StyleBox[\"val\", \"TI\"], StyleBox[\"1\", \"TR\"]}],
        \",\", RowBox[{SubscriptBox[StyleBox[\"key\", \"TI\"], StyleBox[\"2\", \"TR\"]],
        \"→\", SubscriptBox[StyleBox[\"val\", \"TI\"], StyleBox[\"2\", \"TR\"]}], \",\", \"\",
        StyleBox[\"...\", \"TR\"]}], \">\"]}]]) gives the list \\(\\*RowBox[{\"{\",
        RowBox[{RowBox[{StyleBox[\"f\", \"TI\"], \"[\", RowBox[{SubscriptBox[StyleBox[\"key\",
        \"TI\"], StyleBox[\"1\", \"TR\"]}], \",\", \"\", SubscriptBox[StyleBox[\"val\", \"TI\"],
        StyleBox[\"1\", \"TR\"]}], \",\", \"\", RowBox[{StyleBox[\"f\", \"TI\"], \"[\",
        RowBox[{SubscriptBox[StyleBox[\"key\", \"TI\"], StyleBox[\"2\", \"TR\"]}], \",\", \"\",
        SubscriptBox[StyleBox[\"val\", \"TI\"], StyleBox[\"2\", \"TR\"]}], \",\", \"\",
        StyleBox[\"...\", \"TR\"]}], \"}\"]])\". (Note: function backported from v10.1+.)

    "\\(\\*RowBox[{\"KeyValueMap\", \"[\", StyleBox[\"f\", \"TI\"], \"\"]})\" represents
      an operator form of KeyValueMap that can be applied to an expression.";
  KeyValueMap::invak = "The argument `1` is not a valid association";
]

Begin["`Private`"];
If[
  $VersionNumber < 10.1,
  KeyValueMap[f_, assoc_?AssociationQ] := Map[Apply[f], Normal[assoc]];
  KeyValueMap[f_][assoc_?AssociationQ] := KeyValueMap[f, assoc];
  KeyValueMap[f_, assoc_] :=
    "Doesn't display; cf. mm.se/q/29321 for details" /; Message[KeyValueMap::invak, assoc];
]
End[];

```

Dipole transition matrix elements

Default DTME, for a hydrogenic 1s state

```

In[ ]:=
hydrogenicDTME::usage =
  "hydrogenicDTME[p,κ] returns the dipole transition matrix element for a
    1s hydrogenic state of ionization potential  $I_p = -\frac{\kappa^2}{2}$ .

hydrogenicDTME[p,κ,{n,l,m}] returns the dipole transition matrix element for
    an n,l,m hydrogenic state of ground-state ionization potential  $I_p = -\frac{\kappa^2}{2}$ .

```

```

hydrogenicDTME[p,κ,n,l,m] returns the dipole transition matrix element for an
n,l,m hydrogenic state of ground-state ionization potential  $I_p = -\frac{\kappa^2}{2}$ .";

hydrogenicDTMERegularized::usage = "hydrogenicDTMERegularized[p,κ] returns
the dipole transition matrix element for a 1s hydrogenic state of
ionization potential  $I_p = -\frac{\kappa^2}{2}$ , regularized to remove the denominator
of  $1/(p^2 + \kappa^2)^3$ , where the saddle-point solutions are singular.

hydrogenicDTMERegularized[p,κ,{n,l,m}] returns the dipole transition matrix
element for an n,l,m hydrogenic state of ground-state ionization potential
 $I_p = -\frac{\kappa^2}{2}$ , regularized to remove factors of  $(p^2 + \kappa^2)$  from the denominator.

hydrogenicDTMERegularized[p,κ,n,l,m] returns the dipole transition matrix
element for an n,l,m hydrogenic state of ground-state ionization potential
 $I_p = -\frac{\kappa^2}{2}$ , regularized to remove factors of  $(p^2 + \kappa^2)$  from the denominator.";

Begin["`Private`"];

hydrogenicDTME[p_List, κ_] :=  $\frac{8 \, i}{\pi} \frac{\sqrt{2 \, \kappa^5} \, p}{(\text{Total}[p^2] + \kappa^2)^3}$ 

hydrogenicDTME[p_?NumberQ, κ_] :=  $\frac{8 \, i}{\pi} \frac{\sqrt{2 \, \kappa^5} \, p}{(p^2 + \kappa^2)^3}$ 

hydrogenicDTMERegularized[p_List, κ_] :=  $\frac{8 \, i}{\pi} \frac{\sqrt{2 \, \kappa^5} \, p}{1}$ 

hydrogenicDTMERegularized[p_?NumberQ, κ_] :=  $\frac{8 \, i}{\pi} \frac{\sqrt{2 \, \kappa^5} \, p}{1}$ 

End[];

```

For a gaussian orbital


```

In[ ]:= gaussianDTME::usage = "gaussianDTME[p,κ] returns the dipole transition
      matrix element for a gaussian state of characteristic size 1/κ.";
Begin["`Private`"];

gaussianDTME[p_List, κ_] := -i (4 π)3/4 κ-7/2 p Exp[- $\frac{\text{Total}[p^2]}{2 \kappa^2}$ ]

gaussianDTME[p_?NumberQ, κ_] := -i (4 π)3/4 κ-7/2 p Exp[- $\frac{p^2}{2 \kappa^2}$ ]

End[];

```

SolidHarmonics

This function implements the solid harmonic $S_{l,m}(r) = r^l Y_{l,m}(\theta, \phi)$, which is a homogeneous polynomial of degree l , and lends itself much better to symbolic differentiation than explicit spherical harmonics.

Code provided by J.M. at <http://mathematica.stackexchange.com/a/124336/1000> under the WTFPL.

```

In[ ]:= SolidHarmonics::usage =
  "SolidHarmonics[l,m,x,y,z] calculates the solid harmonic  $S_{lm}(x,y,z)=r^l Y_{lm}(x,y,z)$ .

SolidHarmonics[l,m,{x,y,z}] does the same.";
Begin["`Private`"];
SolidHarmonics[λ_Integer, μ_Integer, x_, y_, z_] /; λ ≥ Abs[μ] :=
  Sqrt[ $\frac{2 \lambda + 1}{4 \pi}$ ] Sqrt[ $\frac{\text{Gamma}[\lambda - \text{Abs}[\mu] + 1]}{\text{Gamma}[\lambda + \text{Abs}[\mu] + 1]}$ ] 2-λ (-1)(μ-Abs[μ])/2 x
  If[Rationalize[μ] == 0, 1, (x + Sign[μ] i y)Abs[μ]] x
  Sum[
    (-1)μ+k Binomial[λ, k] Binomial[2 λ - 2 k, λ] Pochhammer[λ - Abs[μ] - 2 k + 1, Abs[μ]] x
    If[TrueQ[Pochhammer[λ - Abs[μ] - 2 k + 1, Abs[μ]] == 0], 1,
      If[Rationalize[k] == 0, 1, (x2 + y2 + z2)k] x If[Rationalize[λ - Abs[μ] - 2 k] == 0, 1, zλ-Abs[μ]-2 k]
    ]
    , {k, 0, Quotient[λ, 2]}]
SolidHarmonics[λ_Integer, μ_Integer, {x_, y_, z_}] /; λ ≥ Abs[μ] := SolidHarmonics[λ, μ, x, y, z]
End[];

```

hydrogenicΨ and hydrogenicY (momentum-space wavefunctions)

This implements the dipole transition matrix element from an arbitrary hydrogenic orbital n, l, m , where the ground-state ionization potential is given by $I_p = \frac{1}{2} \kappa^2$, as described in Luke Chipperfield's PhD thesis (Imperial College London, 2008, p. 52). This code uses partial memoization as in mm.se/q/21782.

```

In[ ]:= hydrogenicΨ::usage =
  "hydrogenicΨ[n,l,m,κ,px,py,pz] calculates the momentum-space wavefunction
  Ψ(p)=⟨p|n,l,m⟩ for a hydrogenic atom with ionization potential κ2/2.

  hydrogenicΨ[n,l,m,κ,{px,py,pz}] calculates the momentum-space wavefunction
  Ψ(p)=⟨p|n,l,m⟩ for a hydrogenic atom with ionization potential κ2/2.";
Begin["`Private`"];
hydrogenicΨ[n_, l_, m_, κκ_, pp_x_, pp_y_, pp_z_] := Block[{κ, px, py, pz},
  hydrogenicΨ[n, l, m, κκ_, px_, py_, pz_] = Simplify[
    -SolidHarmonicsS[l, m, px, py, pz]  $\frac{(-i)^l \pi 2^{2l+4} l!}{(2 \pi \kappa)^{3/2}}$   $\sqrt{\frac{n(n-l-1)!}{(n+l)!}}$ 
     $\frac{\kappa^{l+4}}{(px^2 + py^2 + pz^2 + \kappa^2)^{l+2}}$  GegenbauerC[n-l-1, l+1,  $\frac{px^2 + py^2 + pz^2 - \kappa^2}{px^2 + py^2 + pz^2 + \kappa^2}$ ]
  ];
  hydrogenicΨ[n, l, m, κκ, pp_x, pp_y, pp_z]
];
hydrogenicΨ[n_, l_, m_, κ_, {px_, py_, pz_}] := hydrogenicΨ[n, l, m, κ, px, py, pz];
End[];

```

Regularized version, removing the powers of $p^2 + \kappa^2$ in the denominator, to eliminate poles at the saddle-point momentum $p = i \kappa$.

In[]:=

```

hydrogenicPsiRegularized::usage =
  "hydrogenicYRegularized[n,l,m,κ,px,py,pz] calculates the momentum-space
  wavefunction  $\Psi(p)=\langle p|nlm\rangle$  for a hydrogenic atom with ionization potential  $\kappa^2/2$ ,
  multiplied by  $(p^2+\kappa^2)^{n+1}$  to remove any factors of  $p^2+\kappa^2$  in the denominator.

  hydrogenicYRegularized[n,l,m,κ,{px,py,pz}] calculates the momentum-space
  wavefunction  $\Psi(p)=\langle p|nlm\rangle$  for a hydrogenic atom with ionization potential  $\kappa^2/2$ ,
  multiplied by  $(p^2+\kappa^2)^{n+1}$  to remove any factors of  $p^2+\kappa^2$  in the denominator.";
Begin["`Private`"];
hydrogenicPsiRegularized[n_, l_, m_, κκ_, pp_x_, pp_y_, pp_z_] := Block[{κ, px, py, pz},
  hydrogenicPsiRegularized[n, l, m, κ_, px_, py_, pz_] = Simplify[Cancel[
    -SolidHarmonicsS[l, m, px, py, pz]  $\frac{(-i)^l \pi 2^{2l+4} l!}{(2 \pi \kappa)^{3/2}} \sqrt{\frac{n(n-l-1)!}{(n+l)!}}$ 
     $\kappa^{l+4} (px^2 + py^2 + pz^2 + \kappa^2)^{n-l-1}$  GegenbauerC[n-l-1, l+1,  $\frac{px^2 + py^2 + pz^2 - \kappa^2}{px^2 + py^2 + pz^2 + \kappa^2}$ ]
  ]];
hydrogenicPsiRegularized[n, l, m, κκ, pp_x, pp_y, pp_z]
];
hydrogenicPsiRegularized[n_, l_, m_, κ_, {px_, py_, pz_}] :=
  hydrogenicPsiRegularized[n, l, m, κ, px, py, pz];
End[];

```

Upsilon function, given by $Y(\mathbf{p}) = \left(\frac{1}{2} \mathbf{p}^2 + I_p\right) \Psi(\mathbf{p}) = \frac{1}{2} (\mathbf{p}^2 + \kappa^2) \langle \mathbf{p} | n, l, m \rangle$, which can be used in the form $Y(\mathbf{p} + \mathbf{A}(t'))$ as a replacement for the ionization dipole $\mathbf{d}(\mathbf{p} + \mathbf{A}(t')) \cdot \mathbf{F}(t')$, particularly for cases where the latter is singular but the former is not. (For details cf. arXiv:1304.2413, appendix A.)

```

In[ ]:= hydrogenicY::usage =

    "hydrogenicY[n,l,m,κ,px,py,pz] calculates the Upsilon function  $Y(p)=\frac{1}{2}(-p^2+I_p)\langle p|nlm\rangle$ 
    for a hydrogenic atom with ionization potential  $\kappa^2/2$ .

hydrogenicY[n,l,m,κ,{px,py,pz}] calculates the Upsilon function
 $Y(p)=\frac{1}{2}(-p^2+I_p)\langle p|nlm\rangle$  for a hydrogenic atom with ionization potential  $\kappa^2/2$ .";
Begin["`Private`"];
hydrogenicY[n_, l_, m_, κ_, px_, py_, pz_] :=
     $\frac{1}{2} (px^2 + py^2 + pz^2 + \kappa^2)$  hydrogenicΨ[n, l, m, κ, px, py, pz];
hydrogenicY[n_, l_, m_, κ_, {px_, py_, pz_}] := hydrogenicY[n, l, m, κ, px, py, pz];
End[];

```

hydrogenicDTME for arbitrary states

```

In[ ]:= Begin["`Private`"];
hydrogenicDTME[{ppx_, ppy_, ppz_}, κκ_, n_, l_, m_] := Block[{κ, px, py, pz},
    hydrogenicDTME[{px_, py_, pz_}, κ_, n, l, m] =
        Simplify[Grad[hydrogenicY[n, l, m, κ, px, py, pz], {px, py, pz}]];
    hydrogenicDTME[{ppx, ppy, ppz}, κκ, n, l, m]
];
hydrogenicDTME[{px_, py_, pz_}, κ_, {n_, l_, m_}] := hydrogenicDTME[{px, py, pz}, κ, n, l, m];
End[];

```

Regularized version, removing the powers of $p^2 + \kappa^2$ in the denominator, to eliminate poles at the saddle-point momentum $p = i\kappa$.

```

In[ ]:= Begin["`Private`"];
hydrogenicDTMERegularized[{px_, py_, pz_}, κ_, n_, l_, m_] :=
     $(px^2 + py^2 + pz^2 + \kappa^2)^{n+1}$  hydrogenicDTME[{px, py, pz}, κ, n, l, m];
hydrogenicDTMERegularized[{px_, py_, pz_}, κ_, {n_, l_, m_}] :=
    hydrogenicDTMERegularized[{px, py, pz}, κ, n, l, m];
End[];

```

Various field envelopes

flatTopEnvelope

In[]:=

```
flatTopEnvelope::usage =
  "flatTopEnvelope[ $\omega$ ,num,nRamp] returns a Function object representing a
  flat-top envelope at carrier frequency  $\omega$  lasting a total
  of num cycles and with linear ramps nRamp cycles long.";
Begin["`Private`"];
flatTopEnvelope[ $\omega$ _, num_, nRamp_] := Function[t,
  Piecewise[{{0, t < 0}, {Sin[ $\frac{\omega t}{4 \text{ nRamp}}$ ]2, 0 ≤ t <  $\frac{2 \pi}{\omega} \text{ nRamp}$ }, {1,  $\frac{2 \pi}{\omega} \text{ nRamp}$  ≤ t <  $\frac{2 \pi}{\omega} (\text{num} - \text{nRamp})$ },
    {Sin[ $\frac{\omega (\frac{2 \pi}{\omega} \text{num} - t)}{4 \text{ nRamp}}$ ]2,  $\frac{2 \pi}{\omega} (\text{num} - \text{nRamp})$  ≤ t <  $\frac{2 \pi}{\omega} \text{num}$ }, {0,  $\frac{2 \pi}{\omega} \text{num}$  ≤ t}}]]
End[];
```

cosPowerFlatTop

In[]:=

```
cosPowerFlatTop::usage =
  "cosPowerFlatTop[ $\omega$ ,num,power] returns a Function object representing a
  smooth flat-top envelope of the form  $1 - \text{Cos}(\omega t / 2 \text{ num})^{\text{power}}$ ";
Begin["`Private`"];
cosPowerFlatTop[ $\omega$ _, num_, power_] := Function[t, 1 - Cos[ $\frac{\omega t}{2 \text{ num}}$ ]power]
End[];
```

Field duration standard options

The standard options for the duration of the pulse and the resolution are

```
In[ ]:= PointsPerCycle::usage =
  "PointsPerCycle is a sampling option which specifies the number of sampling
  points per cycle to be used in integrations.";
TotalCycles::usage = "TotalCycles is a sampling option which specifies
  the total number of periods to be integrated over.";
CarrierFrequency::usage = "CarrierFrequency is a sampling option
  which specifies the carrier frequency to be used.";
CarrierFrequency::default = "Warning: no CarrierFrequency was
  specified, using  $\omega=1$  a.u. as the default.";
$DefaultCarrierFrequency::usage = "Default CarrierFrequency to
  use when no explicit option is indicated.";

Protect[PointsPerCycle, TotalCycles, CarrierFrequency];
```

```
In[ ]:= standardOptions = {PointsPerCycle → 90, TotalCycles → 1,
  CarrierFrequency → Automatic, IntegrationPointsPerCycle → Automatic};
$DefaultCarrierFrequency = 0.057;
```

```
In[ ]:= GetCarrierFrequency::usage =
  "GetCarrierFrequency[OptionValue[CarrierFrequency]] returns OptionValue[CarrierFrequency],
  unless it's set to Automatic, in which case it
  returns $DefaultCarrierFrequency and issues a warning.

  GetCarrierFrequency[ $\omega$ ] works for any input.";
Begin["`Private`"];
GetCarrierFrequency[optionvalue_] := If[
  optionvalue === Automatic,
  Message[CarrierFrequency::default, $DefaultCarrierFrequency];
  $DefaultCarrierFrequency,
  optionvalue
]
End[]
```

```
Out[ ]:= RBSFA`Private`
```

PointsPerCycle dictates how many sampling points are used per laser cycle (at frequency CarrierFrequency, of the infrared laser), and it should be at least twice the highest harmonic of interest. The total duration is TotalCycles cycles. CarrierFrequency is the frequency of the fundamental laser, in atomic units.

harmonicOrderAxis

harmonicOrderAxis produces a list that can be used as a harmonic order axis for the given pulse parameters.

The length can be fine-tuned (to match exactly a spectrum, for instance, and get a matrix of the correct shape)

using the correction option, or a TargetLength can be directly specified.

```

In[ ]:= harmonicOrderAxis::usage =
  "harmonicOrderAxis[opt→value] returns a list of frequencies which can be
    used as a frequency axis for Fourier transforms, scaled in units of
    harmonic order, for the provided field duration and sampling options.";
TargetLength::usage = "TargetLength is an option for harmonicOrderAxis which
  specifies the total length required of the resulting list.";
LengthCorrection::usage = "LengthCorrection is an option for harmonicOrderAxis
  which allows for manual correction of the length of the resulting list.";
Protect[LengthCorrection, TargetLength];
Begin["`Private`"];
Options[harmonicOrderAxis] =
  Join[standardOptions, {TargetLength → Automatic, LengthCorrection → 1}];
harmonicOrderAxis::target =
  "Invalid TargetLength option `1`. This must be a positive integer or Automatic.";
harmonicOrderAxis[OptionsPattern[]] :=
  Module[{num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle]},
    Piecewise[{
      {

$$\frac{1}{\text{num}} \text{Range}\left[0., \text{Round}\left[\frac{\text{npp num} + 1}{2.}\right] - 1 + \text{OptionValue}[\text{LengthCorrection}]\right],$$

        OptionValue[TargetLength] === Automatic},
      {

$$\frac{\text{Round}\left[\frac{\text{npp num} + 1}{2.}\right]}{\text{num}} \frac{\text{Range}[0, \text{OptionValue}[\text{TargetLength}] - 1]}{\text{OptionValue}[\text{TargetLength}]},$$

        IntegerQ[OptionValue[TargetLength]] && OptionValue[TargetLength] ≥ 0}
    },
    Message[harmonicOrderAxis::target, OptionValue["TargetLength"]]; Abort[]
  ]
End[];

```

frequencyAxis

frequencyAxis produces a list that can be used as a harmonic order axis for the given pulse parameters. Identical to harmonicOrderAxis but produces a frequency axis (in atomic units) instead.

In[]:=

```

frequencyAxis::usage =
  "frequencyAxis[opt→value] returns a list of frequencies which can be used
  as a frequency axis for Fourier transforms, in atomic units of
  frequency, for the provided field duration and sampling options.";
Begin["`Private`"];
Options[frequencyAxis] = Options[harmonicOrderAxis];
frequencyAxis[options : OptionsPattern[]] :=
  GetCarrierFrequency[OptionValue[CarrierFrequency]] × harmonicOrderAxis[options]
End[];

```

timeAxis

timeAxis produces a list that can be used as a time axis for the given pulse parameters.

Quit


```

In[ ]:=
timeAxis::usage =
  "timeAxis[opt→value] returns a list of times which can be used as a time axis ";
TimeScale::usage = "TimeScale is an option for timeAxis which specifies the units
  the list should use: AtomicUnits by default, or LaserPeriods if required.";
AtomicUnits::usage = "AtomicUnits is a value for the option TimeScale of timeAxis
  which specifies that the times should be in atomic units of time.";
LaserPeriods::usage = "LaserPeriods is a value for the option TimeScale of timeAxis
  which specifies that the times should be in multiples of the carrier laser period.";
Protect[TimeScale, AtomicUnits, LaserPeriods];
Begin["`Private`"];
Options[timeAxis] =
  standardOptions~Join~{TimeScale → AtomicUnits, PointNumberCorrection → 0};
timeAxis::scale =
  "Invalid TimeScale option `1`. Available values are AtomicUnits and LaserPeriods";
timeAxis[OptionsPattern[]] :=
  Block[{T = 2 π / ω, ω = GetCarrierFrequency[OptionValue[CarrierFrequency]],
    num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle]},
    Piecewise[{
      {1, OptionValue[TimeScale] === AtomicUnits},
      {1/T, OptionValue[TimeScale] === LaserPeriods}
    },
    Message[timeAxis::scale, OptionValue[TimeScale]]; Abort[]
  ] × Table[
    t, {t, 0, num  $\frac{2 \pi}{\omega}$ ,  $\frac{\text{num}}{\text{num} \times \text{npp} + \text{OptionValue[PointNumberCorrection]}} \frac{2 \pi}{\omega}$ }
  ]
End[];

```

```
tInit = 0;
```

```
tFinal =  $\frac{2 \pi}{\omega}$  num;
```

```
δt =  $\frac{\text{tFinal} - \text{tInit}}{\text{num} \times \text{npp} + \text{OptionValue[PointNumberCorrection]}}$ ; (*integration and looping timestep*)
```

getSpectrum

getSpectrum takes a time-dependent dipole list and returns its Fourier transform in absolute-value-squared. It takes as options

- pulse parameters ω , TotalCycles and PointsPerCycle,
- a polarization parameter ϵ , which gives an unpolarized spectrum when given False, or polarizes along an ellipticity vector ϵ (this is meant primarily to select right- and left-circularly polarized spectra using $\epsilon = \{1, i\}$ and $\epsilon = \{1, -i\}$ respectively),
- a DifferentiationOrder, which can return the dipole value (default, = 0), velocity (= 1), or acceleration (= 2),
- a power of ω , ω Power, with which to multiply the spectrum before returning it (which should be equivalent to DifferentiationOrder except for pathological cases), and
- a ComplexPart function to apply immediately after differentiation (default is the identity function, but Re, Im, or Abs[#]² & are reasonable choices).

If no option is passed to ω Power and DifferentiationOrder, the pulse parameters do not really affect the output, except by a global factor of TotalCycles.

In[]:=

```
getSpectrum::usage = "getSpectrum[DipoleList] returns the power spectrum of DipoleList.";
Polarization::usage =
  "Polarization is an option for getSpectrum which specifies a polarization
  vector along which to polarize the dipole list. The default,
  Polarization→False, specifies an unpolarized spectrum.";
ComplexPart::usage = "ComplexPart is an option for getSpectrum which
  specifies a function (like Re, Im, or by default #&) which should
  be applied to the dipole list before the spectrum is taken.";
ωPower::usage = "ωPower is an option for getSpectrum which specifies
  a power of frequency which should multiply the spectrum.";
DifferentiationOrder::usage = "DifferentiationOrder is an option for
  getSpectrum which specifies the order to which the dipole
  list should be differentiated before the spectrum is taken.";
Protect[Polarization, ComplexPart, ωPower, DifferentiationOrder];

Begin["`Private`"];
Options[getSpectrum] = {Polarization → False, ComplexPart → (# &),
  ωPower → 0, DifferentiationOrder → 0}~Join~standardOptions;

getSpectrum::diffOrd = "Invalid differentiation order `1`.";
getSpectrum::ωPow = "Invalid ω power `1`.";

getSpectrum[dipoleList_, OptionsPattern[]] := Block[
  {polarizationVector, differentiatedList, depth, dimensions,
    num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle], ω, δt =  $\frac{2 \pi / \omega}{npp}$ 
  },
  polarizationVector =  $\frac{\text{OptionValue[Polarization]}}{\text{Norm[OptionValue[Polarization]]}}$ ;
```

```

differentiatedList = OptionValue[ComplexPart][Piecewise[{
  {dipoleList, OptionValue[DifferentiationOrder] == 0},
  {

$$\frac{1}{2 \delta t} (\text{Most}[\text{Most}[\text{dipoleList}]] - \text{Rest}[\text{Rest}[\text{dipoleList}]]),$$

    OptionValue[DifferentiationOrder] == 1},
  {

$$\frac{1}{\delta t^2} (\text{Most}[\text{Most}[\text{dipoleList}]] - 2 \text{Most}[\text{Rest}[\text{dipoleList}]] + \text{Rest}[\text{Rest}[\text{dipoleList}]]),$$

    OptionValue[DifferentiationOrder] == 2}}],
  Message[getSpectrum::diffOrd, OptionValue[DifferentiationOrder]]; Abort[]
]];

If[NumberQ[OptionValue[ωPower]], Null;, Message[getSpectrum::ωPow, OptionValue[ωPower]];
  Abort[] ];
If[OptionValue[ωPower] ≠ 0, ω = GetCarrierFrequency[OptionValue[CarrierFrequency]], ω = 1];
(*If ωPower==
  0 the value of ω doesn't matter and there's no sense in printing error messages*)

num Table[
  (

$$\frac{\omega}{\text{num}} k$$

2 OptionValue[ωPower]
, {k, 1, Round[

$$\frac{\text{Length}[\text{differentiatedList}]}{2}$$

]}}
] × If[
  OptionValue[Polarization] === False, (*unpolarized spectrum*)
  (*funky depth thing so this can take lists of numbers and lists of vectors,
  of arbitrary length. Makes for easier benchmarking.*)
  depth = Length[Dimensions[dipoleList]];
  dimensions = If[Length[##] > 1, ##[[2]], 1(*##[[1]]*)] &[Dimensions[dipoleList]];
  Sum[Abs[
    Fourier[
      If[depth > 1, Re[differentiatedList[[All, i]], Re[differentiatedList[[All]]]
      , FourierParameters → {-1, 1}
    ][[1 ;; Round[

$$\frac{\text{Length}[\text{differentiatedList}]}{2}$$

]]]]
2
, {i, 1, dimensions}]
, (*polarized spectrum*)
Abs[
  Transpose[Table[
    Fourier[
      Re[differentiatedList[[All, i]]

```

```

      , FourierParameters → {-1, 1}
    ]
    , {i, 1, 2}]]][1 ;; Round[Length[differentiatedList]/ 2]].polarizationVector
  ]^2
]
]
End[];

```

spectrumPlotter

spectrumPlotter takes a spectrum and a list of options and returns a plot of the spectrum. The available options are

- a FrequencyAxis option, which will give the harmonic order as a horizontal axis by default, and an arbitrary scale with any other option,
- all the options of harmonicOrderAxis, which will be passed to the call that makes the horizontal axis, and
- all the options of ListLinePlot, which will be used to format the plot.

```

In[ ]:= spectrumPlotter::usage = "spectrumPlotter[spectrum]
      plots the given spectrum with an appropriate axis in a log10 scale.";
FrequencyAxis::usage = "FrequencyAxis is an option for spectrumPlotter
      which specifies the axis to use.";
Protect[FrequencyAxis];
Begin["`Private`"];
Options[spectrumPlotter] =
  Join[{FrequencyAxis → "HarmonicOrder"}, Options[harmonicOrderAxis], Options[ListLinePlot]];
spectrumPlotter[spectrum_, options : OptionsPattern[]] := ListPlot[
  {Which[
    OptionValue[FrequencyAxis] === "HarmonicOrder",
    harmonicOrderAxis["TargetLength" → Length[spectrum], Sequence @@
      FilterRules[{options}~Join~Options[spectrumPlotter], Options[harmonicOrderAxis]]],
    OptionValue[FrequencyAxis] === "Frequency",
    frequencyAxis["TargetLength" → Length[spectrum], Sequence @@
      FilterRules[{options}~Join~Options[spectrumPlotter], Options[harmonicOrderAxis]]],
    True, Range[Length[spectrum]]
  ],
  Log[10, spectrum]
]
, Sequence @@ FilterRules[{options}, Options[ListLinePlot]]
, Joined → True
, PlotRange → Full
, PlotStyle → Thick
, Frame → True
, Axes → False
, ImageSize → 800
]
End[];

```

biColorSpectrum

biColorSpectrum takes a time-dependent dipole list and produces overlaid plots of the right- and left-circular components of the spectrum, in red and blue respectively. It takes all the options of getSpectrum and spectrumPlotter, which are passed directly to the corresponding calls, as well as the options of Show, which can be used to modify the plot appearance.

Quit

```

In[ ]:= biColorSpectrum::usage =
  "biColorSpectrum[DipoleList] produces a two-colour spectrum of DipoleList,
    separating the two circular polarizations.";
Begin["`Private`"];
Options[biColorSpectrum] = Join[{PlotRange → All}, Options[Show],
  Options[spectrumPlotter], DeleteCases[Options[getSpectrum], Polarization → False]];
biColorSpectrum[dipoleList_, options : OptionsPattern[]] := Show[{
  spectrumPlotter[
    getSpectrum[dipoleList, Polarization → {1, +i},
      Sequence @@ FilterRules[{options}, Options[getSpectrum]]],
    PlotStyle → Red, Sequence @@ FilterRules[{options}, Options[spectrumPlotter]]],
  spectrumPlotter[
    getSpectrum[dipoleList, Polarization → {1, -i},
      Sequence @@ FilterRules[{options}, Options[getSpectrum]]],
    PlotStyle → Blue, Sequence @@ FilterRules[{options}, Options[spectrumPlotter]]]
  }, PlotRange → OptionValue[PlotRange],
  , Sequence @@ FilterRules[{options}, Options[Show]]
]
End[];

```

Various gate functions

Gate functions are used to suppress the contributions of extra-long trajectories with long excursion times, partly to reflect the effect of phase matching but mostly to keep integration times reasonable. They are provided to the main numerical integrator makeDipoleList via its Gate option.

```

In[ ]:= SineSquaredGate::usage =
  "SineSquaredGate[nGateRamp] specifies an integration gate with a sine-squared
    ramp, such that SineSquaredGate[nGateRamp][ωt,nGate]
    has nGate flat periods and nGateRamp ramp periods.";
LinearRampGate::usage = "LinearRampGate[nGateRamp] specifies an integration
  gate with a linear ramp, such that SineSquaredGate[nGateRamp][ωt,nGate]
  has nGate flat periods and nGateRamp ramp periods.";
Begin["`Private`"];
SineSquaredGate[nGateRamp_][ωt_, nGate_] := Piecewise[{{1, ωt ≤ 2 π (nGate - nGateRamp)},
  {Sin[ $\frac{2 \pi nGate - \omega t}{4 nGateRamp}$ ]2, 2 π (nGate - nGateRamp) < ωt ≤ 2 π nGate}, {0, nGate < ωt}}]
LinearRampGate[nGateRamp_][ωt_, nGate_] := Piecewise[{{1, ωt ≤ 2 π (nGate - nGateRamp)},
  {- $\frac{\omega t - 2 \pi (nGate + nGateRamp)}{2 \pi nGateRamp}$ , 2 π (nGate - nGateRamp) < ωt ≤ 2 π nGate}, {0, nGate < ωt}}]
End[];

```

getIonizationPotential

```

In[ ] := getIonizationPotential::usage =
  "getIonizationPotential[Target] returns the ionization potential
    of an atomic target, e.g. \"Hydrogen\", in atomic units.

  getIonizationPotential[Target,q] returns the ionization
    potential of the q-th ion of the specified Target, in atomic units.

  getIonizationPotential[{Target,q}] returns the ionization
    potential of the q-th ion of the specified Target, in atomic units.";
Begin["`Private`"];
getIonizationPotential[Target_, Charge_:0] :=
  UnitConvert[
$$\frac{\text{ElementData}[Target, "IonizationEnergies"][[Charge + 1]]}{\text{Quantity}[1, "AvogadroConstant"] \times \text{Quantity}[1, "Hartrees"]}]
getIonizationPotential[{Target_, Charge_:0}] := getIonizationPotential[Target, Charge]
End[];$$

```

makeDipoleList: main numerical integrator

The main integration function is `makeDipoleList`, and its basic syntax is of the form `makeDipoleList[VectorPotential→A]`. Here the vector potential `A` must be a function object, such that for numeric `t` the construct `A[t]` returns a list of numbers after the appropriate field parameters have been introduced: thus the criterion is that, for a call of the form `makeDipoleList[VectorPotential→A, FieldParameters→pars]`, a call of the form `A[t]//.pars` returns a list of numbers for numeric `t`. To see the available options use `Options[makeDipoleList]`, and to get information on each option use the `?VectorPotential` construct.

```

In[ ] := makeDipoleList::usage = "makeDipoleList[VectorPotential→A]
  calculates the dipole response to the vector potential A.";

VectorPotential::usage =
  "VectorPotential is an option for makeDipole list which specifies the field's vector
  potential. Usage should be VectorPotential→A, where A[t]//.pars must yield a list
  of numbers for numeric t and parameters indicated by FieldParameters→pars.";
VectorPotentialGradient::usage = "VectorPotentialGradient is an option for makeDipole
  list which specifies the gradient of the field's vector potential. Usage should
  be VectorPotentialGradient→GA, where GA[t]//.pars must yield a square matrix of the
  same dimension as the vector potential for numeric t and parameters indicated by
  FieldParameters→pars. The indices must be such that GA[t][[i,j]] returns  $\partial_i A_j[t]$ .";
ElectricField::usage = "ElectricField is an option for makeDipole list which
  specifies an electric field to use in the ionization matrix element,
  in case the time derivative of the vector potential is not desired.
  Usage should be ElectricField→F, where F[t]//.pars must yield a list of

```

```

    numbers for numeric t and parameters indicated by FieldParameters→pars.";
FieldParameters::usage = "FieldParameters is an option for makeDipole list which ";
Preintegrals::usage =
    "Preintegrals is an option for makeDipole list which specifies whether the
    preintegrals of the vector potential should be \"Analytic\" or \"Numeric\".";
ReportingFunction::usage = "ReportingFunction is an option for makeDipole
    list which specifies a function used to report the results, either
    internally (by the default, Identity) or to an external file.";
Gate::usage = "Gate is an option for makeDipole list which specifies the
    integration gate to use. Usage as Gate→g, nGate→n will gate the integral
    at time  $\omega t/\omega$  by  $g[\omega t, n]$ . The default is Gate→SineSquaredGate[1/2].";
nGate::usage = "nGate is an option for makeDipole list which specifies
    the total number of cycles in the integration gate.";
IonizationPotential::usage = "IonizationPotential is an option for makeDipoleList
    which specifies the ionization potential  $I_p$  of the target.";
Target::usage = "Target is an option for makeDipoleList which specifies
    chemical species producing the HHG emission, pulling the ionization
    potential from the Wolfram ElementData curated data set.";
DipoleTransitionMatrixElement::usage = "DipoleTransitionMatrixElement is an option
    for makeDipoleList which specifies a function f to use as the dipole transition
    matrix element, or a pair of functions {fion, frec} to be used separately for the
    ionization and recombination dipoels, to be used in the form  $f[p, \kappa] = f[p, \sqrt{2 I_p}]$ .";
ϵCorrection::usage = "ϵCorrection is an option for makeDipoleList which specifies
    the regularization correction  $\epsilon$ , i.e. as used in the factor  $\frac{1}{(t - t_t + i\epsilon)^{3/2}}$ .";
PointNumberCorrection::usage = "PointNumberCorrection is an option for
    makeDipoleList and timeAxis which specifies an extra number of points
    to be integrated over, which is useful to prevent Indeterminate errors
    when a Piecewise envelope is being differentiated at the boundaries.";
IntegrationPointsPerCycle::usage = "IntegrationPointsPerCycle is an option for
    makeDipoleList which controls the number of points per cycle to use for the
    integration. Set to Automatic, to follow PointsPerCycle, or to an integer.";
RunInParallel::usage = "RunInParallel is an option for makeDipoleList which
    controls whether each RB-SFA instance is parallelized. It accepts False
    as the (Automatic) option, True, to parallelize each instance, or a pair
    of functions {TableCommand, SumCommand} to use for the iteration and
    summing, which could be e.g. {Inactive[ParallelTable], Inactive[Sum]}.";
Simplifier::usage = "Simplifier is an option for makeDipoleList which specifies a
    function to use to simplify the intermediate and final analytical results.";
CheckNumericFields::usage = "CheckNumericFields is an option for makeDipoleList which
    specifies whether to check for numeric values of A[t] and GA[t] for numeric t.";
QuadraticActionTerms::usage = "QuadraticActionTerms is an option for makeDipoleList
    which specifies whether to use quadratic terms in  $\nabla A^2$  in the action.";

```



```
Protect[VectorPotential, VectorPotentialGradient, ElectricField, FieldParameters,
  Preintegrals, ReportingFunction, Gate, nGate, IonizationPotential, Target,  $\epsilon$ Correction,
  PointNumberCorrection, DipoleTransitionMatrixElement, IntegrationPointsPerCycle,
  RunInParallel, Simplifier, CheckNumericFields, QuadraticActionTerms];
```

```
Begin["`Private`"];

```

```
Options[makeDipoleList] = standardOptions~Join~{
  VectorPotential → Automatic, FieldParameters → {},
  VectorPotentialGradient → None, ElectricField → Automatic,
  Preintegrals → "Analytic", ReportingFunction → Identity,
  Gate → SineSquaredGate[1/2], nGate → 3/2,  $\epsilon$ Correction → 0.1,
  IonizationPotential → 0.5,
  Target → Automatic, DipoleTransitionMatrixElement → hydrogenicDTME,
  PointNumberCorrection → 0, Verbose → 0, CheckNumericFields → True,
  RunInParallel → Automatic,
  Simplifier → Identity, QuadraticActionTerms → True
};

```

```
makeDipoleList::gate =
```

```
"The integration gate g provided as Gate→`1` is incorrect. Its usage as
g[`2`,`3`] returns `4` and should return a number.";
```

```
makeDipoleList::pot = "The vector potential A provided as VectorPotential→`1`
  is incorrect or is missing FieldParameters. Its usage as
  A[`2`] returns `3` and should return a list of numbers.";
```

```
makeDipoleList::efield = "The electric field f provided as ElectricField→`1` is
  incorrect or is missing FieldParameters. Its usage as F[`2`] returns `3` and
  should return a list of numbers. Alternatively, use ElectricField→Automatic.";
```

```
makeDipoleList::gradpot = "The vector potential GA provided as
  VectorPotentialGradient→`1` is incorrect or is missing FieldParameters.
  Its usage as GA[`2`] returns `3` and should return a square matrix
  of numbers. Alternatively, use VectorPotentialGradient→None.";
```

```
makeDipoleList::preint = "Wrong Preintegrals option `1`. Valid
  options are \"Analytic\" and \"Numeric\".";
```

```
makeDipoleList::runpar = "Wrong RunInParallel option `1`.";
```

```
makeDipoleList::carrfreq = "Non-numeric option CarrierFrequency `1`.";
```

```
makeDipoleList[OptionsPattern[]] := Block[
```

```
{
```

```
  num = OptionValue[TotalCycles], npp = OptionValue[PointsPerCycle],  $\omega$ ,
  dipoleRec, dipoleIon,  $\kappa$ ,
  A, F, GA, pi, ps, S,
  gate, tGate, setPreintegral,
```

```

tInit, tFinal,  $\delta t$ ,  $\delta t_{\text{int}}$ ,  $\epsilon$  = OptionValue[ $\epsilon$ Correction],
AInt, A2Int, GAIInt, GAdotAInt, AdotGAInt, GAIIntInt,
PScorrectionInt, constCorrectionInt, GAIIntdotGAIntInt, QuadMatrix, q,
simplifier, prefactor, integrand, dipoleList,
TableCommand, SumCommand
},

A[t_] = OptionValue[VectorPotential][t] //. OptionValue[FieldParameters];
If[
  OptionValue[ElectricField] === Automatic, F[t_] = -D[A[t], t];,
  F[t_] = OptionValue[ElectricField][t] //. OptionValue[FieldParameters];
];
GA[t_] = If[
  TrueQ[OptionValue[VectorPotentialGradient] == None],
  Table[0, {Length[A[tInit]]}, {Length[A[tInit]]}],
  OptionValue[VectorPotentialGradient][t] //. OptionValue[FieldParameters]
];

 $\omega$  = GetCarrierFrequency[OptionValue[CarrierFrequency]];
If[! NumberQ[ $\omega$ ] && TrueQ[OptionValue[CheckNumericFields]],
  Message[makeDipoleList::carrfreq,  $\omega$ ];
  Abort[];
tInit = 0;
tFinal =  $\frac{2\pi}{\omega}$  num;
(*looping timestep*)
 $\delta t$  =  $\frac{t_{\text{Final}} - t_{\text{Init}}}{\text{num} \times \text{npp} + \text{OptionValue[PointNumberCorrection]}}$ ;
(*integration timestep*)
 $\delta t_{\text{int}}$  = If[OptionValue[IntegrationPointsPerCycle] === Automatic,  $\delta t$ ,
 $\frac{t_{\text{Final}} - t_{\text{Init}}}{\text{num} \times \text{OptionValue[IntegrationPointsPerCycle]} + \text{OptionValue[PointNumberCorrection]}}$ ];
tGate = OptionValue[nGate]  $\frac{2\pi}{\omega}$ ;
(*Check potential and potential gradient for correctness.*)
(*To do: change logic conditions to constructions on VectorQ[#,NumberQ]& and MatrixQ.*)
If[TrueQ[OptionValue[CheckNumericFields]],
  With[{ $\omega t_{\text{Random}}$  = RandomReal[{ $\omega t_{\text{Init}}$ ,  $\omega t_{\text{Final}}$ }]},
    If[! And @@ (NumberQ /@ A[ $\omega t_{\text{Random}}$  /  $\omega$ )],
      Message[makeDipoleList::pot, OptionValue[VectorPotential],  $\omega t_{\text{Random}}$ , A[ $\omega t_{\text{Random}}$ ]];
      Abort[];
    If[! And @@ (NumberQ /@ Flatten[GA[ $\omega t_{\text{Random}}$  /  $\omega$ ]]), Message[makeDipoleList::gradpot,
      OptionValue[VectorPotentialGradient],  $\omega t_{\text{Random}}$ , GA[ $\omega t_{\text{Random}}$ ]];

```

```

Abort[]];
If[! And @@ (NumberQ /@ F[ωtRandom / ω]), Message[makeDipoleList::efield,
  OptionValue[ElectricField], ωtRandom, F[ωtRandom]];
Abort[]];
];

gate[ωr_] := OptionValue[Gate][ωr, OptionValue[nGate]];
With[{ωtRandom = RandomReal[{ω tInit, ω tFinal}]},
  If[! TrueQ[NumberQ[gate[ωtRandom]]],
    Message[makeDipoleList::gate,
      OptionValue[Gate], ωtRandom, OptionValue[nGate], gate[ωtRandom]];
    Abort[]];
];

(*Target setup*)
Which[
  OptionValue[Target] === Automatic, κ =  $\sqrt{2 \text{OptionValue[IonizationPotential]}}$ ,
  True, κ =  $\sqrt{2 \text{getIonizationPotential[OptionValue[Target]}}$ 
];
With[{dim = Length[A[RandomReal[{ω tInit, ω tFinal}]]]},
  (*Explicit conjugation of the
  recombination matrix element to keep the integrand analytic.*)
  Which[
    Head[OptionValue[DipoleTransitionMatrixElement]] === List,
    dipoleIon[{p1_, p2_, p3_}[1 ;; dim], κκ_] =
      First[OptionValue[DipoleTransitionMatrixElement]][{p1, p2, p3}[1 ;; dim], κκ];
    dipoleRec[{p1_, p2_, p3_}[1 ;; dim], κκ_] = Assuming[{{p1, p2, p3, κκ} ∈ Reals}, Simplify[
      Conjugate[Last[OptionValue[DipoleTransitionMatrixElement]][{p1, p2, p3}[1 ;; dim], κκ]]
    ]];
    , True,
    dipoleIon[{p1_, p2_, p3_}[1 ;; dim], κκ_] =
      OptionValue[DipoleTransitionMatrixElement][{p1, p2, p3}[1 ;; dim], κκ];
    dipoleRec[{p1_, p2_, p3_}[1 ;; dim], κκ_] = Assuming[{{p1, p2, p3, κκ} ∈ Reals}, Simplify[
      Conjugate[OptionValue[DipoleTransitionMatrixElement][{p1, p2, p3}[1 ;; dim], κκ]]
    ]];
  ];
];

simplifier = OptionValue[Simplifier];
q = Boole[TrueQ[OptionValue[QuadraticActionTerms]]];

setPreintegral[integralVariable_, preintegrand_,

```

```

dimensions_, integrateWithoutGradient_, parametric_] := Which[
OptionValue[VectorPotentialGradient] != None || TrueQ[integrateWithoutGradient],
(*Vector potential gradient specified,
or integral variable does not depend on it, so integrate*)
Which[
OptionValue[Preintegrals] == "Analytic",
integralVariable[t_, tt_] =
simplifier[(((# /. { $\tau \rightarrow t$ }) - (# /. { $\tau \rightarrow tt$ })) & [Integrate[preintegrand[ $\tau$ , tt],  $\tau$ ]]];

, OptionValue[Preintegrals] == "Numeric",
Which[
TrueQ[Not[parametric]],
Block[{innerVariable},
integralVariable[t_, tt_] = (innerVariable[t] - innerVariable[tt] /. First[
NDSolve[{innerVariable'[ $\tau$ ] == preintegrand[ $\tau$ ,
innerVariable[tInit] == ConstantArray[0, dimensions]],
innerVariable, { $\tau$ , tInit, tFinal}, MaxStepSize  $\rightarrow$  0.25 /  $\omega$ ]
]);
, True,
Block[{matrixpreintegrand, innerVariable, rpre},
matrixpreintegrand[indices_, t_?NumericQ, tt_?NumericQ] :=
preintegrand[t, tt][[## & @@ indices]];
integralVariable[t_, tt_] = Array[(
innerVariable[##][t - tt, tt] /. First@NDSolve[{
D[innerVariable[##][rpre, tt], rpre] ==
Piecewise[{{matrixpreintegrand[##, tt + rpre, tt], tt + rpre  $\leq$  tFinal}}, 0],
innerVariable[##][0, tt] == 0
}, innerVariable[##]
, {rpre, 0, tFinal - tInit}, {tt, tInit, tFinal}
, MaxStepSize  $\rightarrow$  0.25 /  $\omega$ 
]
)&, dimensions];
];
];
, OptionValue[VectorPotentialGradient] === None, (*Vector potential gradient has not been
specified, and integral variable depends on it, so return appropriate zero matrix*)
integralVariable[t_] = ConstantArray[0, dimensions];
integralVariable[t_, tt_] = ConstantArray[0, dimensions];
];

Apply[setPreintegral,

```



```

    ps[t, tt].PScorrectionInt[t, tt]+constCorrectionInt[t, tt]
  )
];

prefactor[t_, τ_] :=

$$\tilde{\epsilon} \left( \frac{2\pi}{\epsilon + \tilde{\epsilon} \tau} \right)^{3/2} \text{dipoleRec}[\text{pi}[ps[t, t - \tau], t, t - \tau], \kappa] \times \text{dipoleIon}[\text{pi}[ps[t, t - \tau], t - \tau, t - \tau], \kappa].F[t - \tau];$$

integrand[t_, τ_] := prefactor[t, τ] Exp[- $\tilde{\epsilon}$  S[t, t - τ]] gate[ω τ];

(*Debugging constructs. Verbose→
  1 prints information about the internal functions. Verbose→2 returns all the relevant
  internal functions and stops. Verbose→3 for quantum-orbit constructs.*)
Which[
  OptionValue[Verbose] == 1, Information /@ {A, GA, ps, pi, S, AInt, A2Int, GAInt, GAdotAInt,
    AdotGAInt, GAIntInt, PScorrectionInt, constCorrectionInt, GAIntdotGAIntInt},
  OptionValue[Verbose] == 2, Return[With[{t = Symbol["t"], tt = Symbol["tt"], τ = Symbol["τ"],
    p = {Symbol["p1"], Symbol["p2"], Symbol["p3"]}}][1 ;; Length[A[ω tInit]]],
    {A[t], GA[t], ps[t, tt], pi[p, t, tt], S[t, tt], AInt[t, tt], A2Int[t, tt], GAInt[t, tt],
    GAdotAInt[t, tt], AdotGAInt[t, tt], GAIntInt[t, tt], PScorrectionInt[t, tt],
    constCorrectionInt[t, tt], GAIntdotGAIntInt[t, tt], QuadMatrix[t, tt], integrand[t, τ]}]],
  OptionValue[Verbose] == 3,
  Return[{
    Function[Evaluate[prefactor[##1, ##1 - ##2]]], Function[Evaluate[S[##1, ##2]]]
  }]
];

(*Single-run parallelization*)
Which[
  OptionValue[RunInParallel] === Automatic ||
    OptionValue[RunInParallel] === False, TableCommand = Table;
  SumCommand = Sum;;
  OptionValue[RunInParallel] === True, TableCommand = ParallelTable;
  SumCommand = Sum;;
  True, TableCommand = OptionValue[RunInParallel][[1]];
  SumCommand = OptionValue[RunInParallel][[2]];
];

(*Numerical integration loop*)
dipoleList = Table[
  OptionValue[ReportingFunction][
    δtintSum[
      integrand[t, τ]
    ], {τ, 0, If[OptionValue[Preintegrals] == "Analytic", tGate, Min[t - tInit, tGate]], δtint}]

```

```

]
, {t, tInit, tFinal,  $\delta t$ }
];
dipoleList

]
End[];

```

Quantum orbit functions suite

Complex root finder

This section implements a routine for solving contains subroutines for the numerical solution of multiple simultaneous complex-valued transcendental equations, essentially by using the Newton's-method solver implemented in FindRoot, and seeding it multiple times with a random (or quasi-random) seed from a box. This code has been taken from the EPToolbox package, which is located and better documented at <https://github.com/episanty/EP-Toolbox>, and it is also documented in <http://mathematica.stackexchange.com/a/57821/1000>.

In[]:=

```

FindComplexRoots::usage =
  "FindComplexRoots[e1==e2, {z, zmin, zmax}] attempts to find complex roots of
  the equation e1==e2 in the complex rectangle with corners zmin and zmax.

FindComplexRoots[{e1==e2, e3==e4, ...}, {z1, z1min, z1max}, {z2, z2min, z2max}, ...]
  attempts to find complex roots of the given system of equations in the
  multidimensional complex rectangle with corners z1min, z1max, z2min, z2max, ...";
Seeds::usage = "Seeds is an option for FindComplexRoots which determines how many
  initial seeds are used to attempt to find roots of the given equation.";
SeedGenerator::usage = "SeedGenerator is an option for FindComplexRoots which determines
  the function used to generate the seeds for the internal FindRoot call. Its
  value can be RandomComplex, RandomNiederreiterComplexes, RandomSobolComplexes,
  DeterministicComplexGrid, or any function f such that f[{zmin, zmax}, n]
  returns n complex numbers in the rectangle with corners zmin and zmax.";

Options[FindComplexRoots] = Join[Options[FindRoot],
  {Seeds -> 50, SeedGenerator -> RandomComplex, Tolerance -> Automatic, Verbose -> False}];
SyntaxInformation[FindComplexRoots] = {"ArgumentsPattern" -> {_, {_, _, _}, OptionsPattern[]},
  "LocalVariables" -> {"Table", {2,  $\infty$ }}};
FindComplexRoots::seeds = "Value of option Seeds -> `1` is not a positive integer.";
FindComplexRoots::tol =
  "Value of option Tolerance -> `1` is not Automatic or a number in [0,  $\infty$ ).";
$MessageGroups = Join[$MessageGroups, {"FindComplexRoots" -> {FindRoot::lstol}}];

Protect[Seeds];
Protect[SeedGenerator];

```

In[*]:=

```

SetTolerances::usage =
  "SetTolerances[tolerance,length] produces a list of the given length with
  the specified tolerance, which may be a number or a list of numbers.\n
SetTolerances[tolerance,length,workingPrecision] allows a fallback to a specified
  workingPrecision in case the given tolerance fails to be numeric.";
Begin["`Private`"];
SetTolerances[tolerance_, length_, workingPrecision_:$MachinePrecision] := Which[
  ListQ[tolerance], tolerance,
  True, ConstantArray[
    Which[
      NumberQ[tolerance], tolerance,
      True, 10^If[NumberQ[workingPrecision], 2-workingPrecision, 2-$MachinePrecision]
    ], length]
]
End[];

```

In[*]:=

```

Begin["`Private`"];
FindComplexRoots[equations_List, domainSpecifiers__, ops : OptionsPattern[]] :=
  Block[{seeds, tolerances},
    If[! IntegerQ[Rationalize[OptionValue[Seeds]]] || OptionValue[Seeds] ≤ 0,
      Message[FindComplexRoots::seeds, OptionValue[Seeds]];
    If[! (OptionValue[Tolerance] === Automatic || OptionValue[Tolerance] ≥ 0),
      Message[FindComplexRoots::tol, OptionValue[Seeds]];

    seeds = OptionValue[SeedGenerator][{domainSpecifiers}[All, {2, 3}], OptionValue[Seeds]];
    tolerances = SetTolerances[OptionValue[Tolerance],
      Length[{domainSpecifiers}], OptionValue[WorkingPrecision]];

    If[OptionValue[Verbose], Hold[], Hold[FindRoot::lstol]] /. {
      Hold[messageSequence___] := Quiet[
        DeleteDuplicates[
          Select[
            Check[
              FindRoot[
                equations
              , Evaluate[Sequence @@
                Table[{domainSpecifiers}[j, 1], #][j]], {j, Length[{domainSpecifiers}]]]]
            , Evaluate[Sequence @@ FilterRules[{ops}, Options[FindRoot]]]
          ],
        ### &[]
      ] & /@ seeds,
    Function[

```



```

repList,
ReplaceAll[
  Evaluate[And @@ Table[
    And[
      Re[{domainSpecifiers}[[j, 2]] ≤ Re[
        {domainSpecifiers}[[j, 1]] ≤ Re[{domainSpecifiers}[[j, 3]],
      Im[{domainSpecifiers}[[j, 2]] ≤ Im[{domainSpecifiers}[[j, 1]] ≤ Im[
        {domainSpecifiers}[[j, 3]]
      ]
    ], {j, Length[{domainSpecifiers}]}]]
  ], repList]
],
Function[{repList1, repList2},
  And @@ Table[
    Abs[({domainSpecifiers}[[j, 1]] /. repList1) -
      ({domainSpecifiers}[[j, 1]] /. repList2)] < tolerances[[j]]
    , {j, Length[{domainSpecifiers}]}]
  ]
], {messageSequence}}]
]
FindComplexRoots[e1_ == e2_, {z_, zmin_, zmax_}, ops : OptionsPattern[] :=
  FindComplexRoots[{e1 == e2}, {z, zmin, zmax}, ops]
End[];

```

Quasirandom number generators

This section implements quasirandom number generators for use with FindComplexRoots. As above, this code has been taken from the EPTtoolbox package, which is located and better documented at <https://github.com/episanty/EPTtoolbox>, and it is also documented in <http://mathematica.stackexchange.com/a/57821/1000>.

RandomSobolComplexes

```

In[ ]:= RandomSobolComplexes::usage =
  "RandomSobolComplexes[{zmin, zmax}, n] generates a low-discrepancy Sobol sequence of
  n quasirandom complex numbers in the rectangle with corners zmin and zmax.

  RandomSobolComplexes[{{z1min,z1max},{z2min,z2max},...},n] generates a
  low-discrepancy Sobol sequence of n quasirandom complex numbers in the
  multi-dimensional rectangle with corners {z1min,z1max},{z2min,z2max},...";

```

```

In[ ]:=
Begin["`Private`"];
RandomSobolComplexes[pairsList__, number_] := Map[
  Function[randomsList,
    pairsList[[All, 1]] + Complex @@@ Times[
      ReIm[pairsList[[All, 2]] - pairsList[[All, 1]],
      randomsList
    ]
  ],
  BlockRandom[
    SeedRandom[Method → {"MKL", Method → {"Sobol", "Dimension" → 2 Length[pairsList]}]];
    SeedRandom[];
    RandomReal[{0, 1}, {number, Length[pairsList], 2}]
  ]
]
RandomSobolComplexes[{zmin_?NumericQ, zmax_?NumericQ}, number_] :=
  RandomSobolComplexes[{{zmin, zmax}}, number][[All, 1]]
End[];

```

RandomNiederreiterComplexes

```

In[ ]:=
RandomNiederreiterComplexes::usage =
  "RandomNiederreiterComplexes[{zmin, zmax}, n] generates a low-discrepancy
  Niederreiter sequence of n quasirandom complex
  numbers in the rectangle with corners zmin and zmax.

  RandomNiederreiterComplexes[{{z1min,z1max},{z2min,z2max},...},n] generates a
  low-discrepancy Niederreiter sequence of n quasirandom complex numbers in
  the multi-dimensional rectangle with corners {z1min,z1max},{z2min,z2max},...";

```

In[]:=

```

Begin["`Private`"];
RandomNiederreiterComplexes[pairsList__, number_] := Map[
  Function[randomsList,
    pairsList[[All, 1]] + Complex @@@ Times[
      ReIm[pairsList[[All, 2]] - pairsList[[All, 1]],
      randomsList
    ]
  ],
  BlockRandom[
    SeedRandom[Method → {"MKL", Method → {"Niederreiter", "Dimension" → 2 Length[pairsList]}]];
    SeedRandom[];
    RandomReal[{0, 1}, {number, Length[pairsList], 2}]
  ]
]
RandomNiederreiterComplexes[{zmin_?NumericQ, zmax_?NumericQ}, number_] :=
  RandomNiederreiterComplexes[{{zmin, zmax}}, number][[All, 1]]
End[];

```

DeterministicComplexGrid

In[]:=

```

DeterministicComplexGrid::usage =
  "DeterministicComplexGrid[{zmin, zmax}, n] generates a grid of about n equally
    spaced complex numbers in the rectangle with corners zmin and zmax.

DeterministicComplexGrid[{{z1min,z1max},{z2min,z2max},...},n]
    generates a regular grid of about n equally spaced complex numbers in the
    multi-dimensional rectangle with corners {z1min,z1max},{z2min,z2max},...";

```

```

In[ ] :=
Begin["`Private`"];
DeterministicComplexGrid[pairsList_, number_] :=
Block[{sep, separationsList, gridPointBasis, k},
  sep = NestWhile[0.99 # &, Min[Flatten[ReIm[pairsList[[All, 2]] - pairsList[[All, 1]]]],
    Floor[Flatten[ReIm[pairsList[[All, 2]] - pairsList[[All, 1]]], 0.99 #]
    Times @@  $\frac{\text{Floor}[\text{Flatten}[\text{ReIm}[\text{pairsList}[[\text{All}, 2]] - \text{pairsList}[[\text{All}, 1]]], 0.99 \#]}{0.99 \#} \leq \text{number} \&];
  separationsList = Round[ $\frac{\text{Floor}[\text{Flatten}[\text{ReIm}[\text{pairsList}[[\text{All}, 2]] - \text{pairsList}[[\text{All}, 1]]], \text{sep}]}{\text{sep}}$ ];
  gridPointBasis = MapThread[
    Function[{l, n}, Range[l[[1]], l[[2]],  $\frac{l[[2]] - l[[1]]}{n + 1}$ ][[2 ;; -2]],
    {Flatten[Transpose[ReIm[pairsList], {1, 3, 2}], 1], separationsList}
  ];
  Flatten[Table[
    Table[k[2 j - 1] + I k[2 j], {j, 1, Length[pairsList]}],
    Evaluate[Sequence @@ Table[{k[j], gridPointBasis[[j]]}, {j, 1, 2 Length[pairsList]}]]
  ], Evaluate[Range[1, 2 Length[pairsList]]]]
];
DeterministicComplexGrid[{zmin_?NumericQ, zmax_?NumericQ}, number_] :=
  DeterministicComplexGrid[{{zmin, zmax}}, number][[All, 1]]
End[];$ 
```

RandomComplex

Updating RandomComplex to handle input of the form RandomComplex[{{0, 1+I},{2, 3+I}},n].

```

In[ ] :=
Begin["`Private`"];
Unprotect[RandomComplex];
RandomComplex[{range1_List, moreRanges___}, number_] :=
  Transpose[RandomComplex[#, number] & /@ {range1, moreRanges}]
Protect[RandomComplex];
End[];

```

The following code places this redefinition as an initialization code for any parallelized subkernels that may get launched later (cf. [mm.se/q/131856](https://math.stackexchange.com/q/131856)). This version, in addition, checks whether there is already any code in \$InitCode and, if there is, it appends its own code there.

```

In[ ] := Parallelize;
If[Head[Parallel`Developer`$InitCode] != Hold,
  Parallel`Developer`$InitCode = Hold[]
];
Parallel`Developer`$InitCode = Join[
  Parallel`Developer`$InitCode,
  Hold[
    Unprotect[RandomComplex];
    RandomComplex[{Private`range1_List, Private`moreRanges___}, Private`number_] :=
      Transpose[RandomComplex[#, Private`number] & /@ {Private`range1, Private`moreRanges}];
    Protect[RandomComplex];
  ]
];

```

ConstrainedDerivative

In some situations, one can require solving the t' saddle-point equation $\frac{\partial}{\partial t'} S(t, t') = 0$ first to find a saddle point $t' = t_s'(t)$ and an action $S(t) = S(t, t_s'(t))$, and then find the derivatives of this function with respect to t . This can be done by differentiating the defining equation $\frac{\partial}{\partial t'} S(t, t_s'(t)) \equiv 0$ with respect to t to find the relation obeyed by $\frac{dt_s'}{dt}$, in the form $\frac{\partial^2 S}{\partial t' \partial t}(t, t_s'(t)) + \frac{dt_s'}{dt}(t) \frac{\partial^2 S}{(\partial t')^2}(t, t_s'(t)) \equiv 0$, and using

```

ConstrainedDerivative::usage =
  "ConstrainedDerivative[n][f][t,tt] calculates the nth derivative of f[t,tt]
  with respect to t under the constraint that Derivative[0,1][f][t,tt]≡0.";

Begin["`Private`"];
ConstrainedDerivative[n_][F_][te_, tte_] := Block[{f, tts, t, tt},
  ConstrainedDerivative[n][f][t_, tt_] = Nest[
    Function[
      Simplify[
        D[#, /. {tt → tts[t]}, t] /.
          {Derivative[0, 1][f][t, tts[t]] → 0, tts'[t] → -
            
$$\frac{\text{Derivative}[1, 1][f][t, \text{tts}[t]]}{\text{Derivative}[0, 2][f][t, \text{tts}[t]]}$$

        }
      ] /. {tts[t] → tt}
    ]
  ], f[t, tt], n];
ConstrainedDerivative[n][F][te, tte]
]
End[];

```

GetSaddlePoints

In[*]:=

```

GetSaddlePoints::usage =
  "GetSaddlePoints[Ω,S,{tmin,tmax},{rmin,rmax}] finds a list of solutions
  {t,τ} of the HHG temporal saddle-point equations at harmonic energy
  Ω for action S, in the range {tmin, tmax} of recombination time
  and {rmin, rmax} of excursion time, where both ranges should be the
  lower-left and upper-right corners of rectangles in the complex plane.

  GetSaddlePoints[ΩRange,S,{tmin,tmax},{rmin,rmax}] finds
  solutions of the HHG temporal saddle-point equations for a range
  of harmonic energies ΩRange, and returns an Association with each
  harmonic energy Ω indexing a list of saddle-point solution pairs {t,τ}.

  GetSaddlePoints[Ωspec,S,{{tmin1,tmax1},{rmin1,rmax1}},{{tmin2,tmax2},{rmin2,rmax2}},...]
  uses multiple time domains and combines the solutions.

  GetSaddlePoints[Ωspec,S,{{urange,vrange},...},IndependentVariables→{u,v}] uses the explicit
  independent variables u and v to solve the equations and over the given
  ranges, where u and v can be any of "RecombinationTime", "IonizationTime"
  and "ExcursionTime", or their shorthands "t", "tt" and "τ" resp.";
SortingFunction::usage = "SortingFunction is an option of GetSaddlePoints

```

```

which sets a function f, to be used as f[t,τ,S,Ω], to be
used to sort the solutions, or a list of such functions.";
SelectionFunction::usage = "SelectionFunction is an option of GetSaddlePoints
that sets a function f, to be used as f[t,τ,S,Ω],
such that roots are only kept if f returns True.";
IndependentVariables::usage = "IndependentVariables is an option for
GetSaddlePoints that specifies the two independent variables, out of
\"RecombinationTime\", \"IonizationTime\" and \"ExcursionTime\" (or their
shorthands \"t\", \"tt\" and \"τ\", respectively), to be used in solving
the saddle-point equations, and which range over the given regions.";

FiniteDifference::usage =
  "FiniteDifference is a value for the option Jacobian of FindRoot, FindComplexRoots,
  GetSaddlePoints, and related functions, which specifies that the Jacobian at
  each step should be evaluated using numerical finite difference procedures.";

GetSaddlePoints::error = "Errors encountered for harmonic energy Ω=`1`.";

Begin["`Private`"];
Options[GetSaddlePoints] = Join[{SortingFunction → (##2 &), SelectionFunction → (True &),
  IndependentVariables → {"RecombinationTime", "ExcursionTime"}}, Options[FindComplexRoots]];
Protect[SortingFunction, SelectionFunction, IndependentVariables, FiniteDifference];

GetSaddlePoints[Ωspec_, S_, {tmin_, tmax_}, {τmin_, τmax_}, options : OptionsPattern[] :=
  GetSaddlePoints[Ωspec, S, {{{tmin, tmax}, {τmin, τmax}}}, options]

GetSaddlePoints[Ω_, S_, timeRanges_, options : OptionsPattern[] :=
  Block[{equations, roots, t = Symbol["t"], tt = Symbol["tt"],
    τ = Symbol["τ"], indVars, depVar, depVarRule, tolerances},
    tolerances = SetTolerances[OptionValue[Tolerance], 2, OptionValue[WorkingPrecision]];
    indVars = OptionValue[IndependentVariables] /.
      {"RecombinationTime" → "t", "ExcursionTime" → "τ", "IonizationTime" → "tt"};
    depVar = First[DeleteCases[{"t", "τ", "tt"}, Alternatives @@ indVars]];
    depVarRule = depVar /. {"tt" → {tt → t - τ}, "t" → {t → tt + τ}, "τ" → {τ → t - tt}};
    equations = {D[S[t, tt], t] == Ω, D[S[t, tt], tt] == 0} /. depVarRule;

    SortBy[
      DeleteDuplicates[
        Flatten[Table[
          Select[
            Check[
              roots = ({t, τ} /. depVarRule) /. (FindComplexRoots[
                equations
                , Evaluate[Sequence[{Symbol[indVars[[1]], range[[1, 1]],
                  range[[1, 2]], {Symbol[indVars[[2]], range[[2, 1]], range[[2, 2]]}]]]]]]]]

```

```

        , Evaluate[Sequence @@ FilterRules[{options}, Options[FindComplexRoots]]]
        , SeedGenerator → RandomSobolComplexes
        , Seeds → 50
      ] /. {{t} → ({t,  $\tau$ } /. depVarRule) → {}} (*to deal with empty results*)
      , Message[GetSaddlePoints::error,  $\Omega$ ]; roots
    ]
    , Function[timesPair, OptionValue[SelectionFunction][timesPair[[1], timesPair[[2]], S,  $\Omega$ ]]
  ]
  , {range, timeRanges}], 1]
, Function[{timesPair1, timesPair2},
  And @@ Thread[Abs[timesPair1 - timesPair2] < tolerances] ]
]
, If[
  ListQ[OptionValue[SortingFunction]],
  Table[Function[timesPair, f[timesPair[[1], timesPair[[2]], S,  $\Omega$ ]],
    {f, OptionValue[SortingFunction]}],
  Function[timesPair, OptionValue[SortingFunction][timesPair[[1], timesPair[[2]], S,  $\Omega$ ]]
]
]
]
]
GetSaddlePoints[ $\Omega$ Range_List, S_, timeRanges_, options : OptionsPattern[]] :=
Association[ParallelTable[
   $\Omega$  → GetSaddlePoints[ $\Omega$ , S, timeRanges, options]
, { $\Omega$ , Sort[ $\Omega$ Range]}]]

End[];

```

GetSaddlesFromSeeds

In[] :=

```

GetSaddlesFromSeeds::usage =
  "GetSaddlesFromSeeds[{{t1,  $\tau$ 1}, {t2,  $\tau$ 2}, ...},  $\Omega$ , S] finds a list of solutions
  {t,  $\tau$ } of the HHG temporal saddle-point equations at harmonic energy
   $\Omega$  for action S, using the given {ti,  $\tau$ i} as seeds for the process.

  GetSaddlesFromSeeds[<|  $\Omega$ 1 → {{t11,  $\tau$ 11}, {t12,  $\tau$ 12}, ...},  $\Omega$ 2 → {{t21,  $\tau$ 21}, {t22,  $\tau$ 22}, ...}, ...|>,  $\Omega$ , S] finds
  solutions of the HHG temporal saddle-point equations, using the seeds list from
  the  $\Omega_i$  that's closest to  $\Omega$ , or as specified by the value of KeyChooserFunction.

  GetSaddlesFromSeeds[seeds, { $\Omega$ 1,  $\Omega$ 2, ...}, S] iterates over the given set of harmonic energies.";

SeedsChooserFunction::usage =
  "SeedsChooserFunction is an option for GetSaddlesFromSeeds that specifies a
  function f (set by default to Nearest) that, when used as f[{ $\Omega$ 1,  $\Omega$ 2, ...},  $\Omega$ ],
  should return the indices { $\Omega_i$ ,  $\Omega_j$ , ...} corresponding to the seed sets

```



```

{{{ti1,  $\tau_{i1}$ }, ...}, {{tj1,  $\tau_{j1}$ }, ...}} to be used to solve the HHG saddle-point equations.";
RecalculateRoots::usage = "RecalculateRoots is an option for GetSaddlesFromSeeds that
specifies whether to re-solve the saddle-point equations if the given harmonic
energy  $\Omega$  is among the set of keys of the given seeds association. The default is
False, which is appropriate for S being the same action used to find the seeds,
in which case setting RecalculateRoots→True will produce multiple FindRoot
errors. If using a different action than used to find the seeds, set to True.";

GetSaddlesFromSeeds::error = "Errors encountered for harmonic energy  $\Omega$ =`1`.";
GetSaddlesFromSeeds::norecalc =
  "Skipping re-calculation of roots at harmonic energy `1` since
  it is already in the key set of the given seeds association. To
  run the calculation for this case set RecalculateRoots to True.";

Begin["`Private`"];
Options[GetSaddlesFromSeeds] =
  Join[{RecalculateRoots → False, SeedsChooserFunction → Nearest}, Options[GetSaddlePoints]];
Protect[SeedsChooserFunction, RecalculateRoots];

GetSaddlesFromSeeds[seedsSpec_,  $\Omega$ Range_List, S_, options : OptionsPattern[]] :=
  Association[ParallelTable[
     $\Omega$  → GetSaddlesFromSeeds[seedsSpec,  $\Omega$ , S, options]
  , { $\Omega$ , Sort[ $\Omega$ Range]}]]

GetSaddlesFromSeeds[seedsAssociation_Association,  $\Omega$ _, S_, options : OptionsPattern[]] :=
  With[{keys = OptionValue[SeedsChooserFunction][Keys[seedsAssociation],  $\Omega$ ]},
    If[MemberQ[keys,  $\Omega$ ] && TrueQ[! OptionValue[RecalculateRoots]],
      Message[GetSaddlesFromSeeds::norecalc,  $\Omega$ ];
      Return[seedsAssociation[ $\Omega$ ]];
    GetSaddlesFromSeeds[Flatten[Values[seedsAssociation[Key /@ keys]], 1],  $\Omega$ , S, options]
  ]

GetSaddlesFromSeeds[seedsList_List,  $\Omega$ ?NumberQ, S_, options : OptionsPattern[]] := Block[
  {equations, roots, t = Symbol["t"], tt = Symbol["tt"],
     $\tau$  = Symbol[" $\tau$ "], indVars, depVar, depVarRule, fullSeedVars, tolerances},
  tolerances = SetTolerances[OptionValue[Tolerance], 2, OptionValue[WorkingPrecision]];
  indVars = OptionValue[IndependentVariables] /.
    {"RecombinationTime" → "t", "ExcursionTime" → " $\tau$ ", "IonizationTime" → "tt"};
  depVar = First[DeleteCases[{"t", " $\tau$ ", "tt"}, Alternatives @@ indVars]];
  depVarRule = depVar /. {"tt" → {tt → t -  $\tau$ }, "t" → {t → tt +  $\tau$ }, " $\tau$ " → { $\tau$  → t - tt}};
  fullSeedVars[seed_] := <|"t" → seed[[1]], " $\tau$ " → seed[[2]], "tt" → seed[[1]] - seed[[2]]|>;
  equations = {D[S[t, tt], t] ==  $\Omega$ , D[S[t, tt], tt] == 0} /. depVarRule;

```

```

SortBy[
  DeleteDuplicates[
    Select[
      Table[
        Check[
          roots = ({t,  $\tau$ } /. depVarRule) /. (
            FindRoot[
              equations
              , {Symbol[ $\Omega$ ], fullSeedVars[seed][ $\Omega$ ] & /@ indVars
              , Evaluate[Sequence @@ FilterRules[{options}, Options[FindRoot]]]
            ]
          /. {{}}  $\rightarrow$  ({t,  $\tau$ } /. depVarRule)  $\rightarrow$  {}}))
          , Message[GetSaddlesFromSeeds::error,  $\Omega$ ]; roots
        ]
        , {seed, seedsList}]
      , Function[timesPair, OptionValue[SelectionFunction][timesPair[[1], timesPair[[2]], S,  $\Omega$ ]]
    ]
    , Function[{timesPair1, timesPair2},
      And @@ Thread[Abs[timesPair1 - timesPair2] < tolerances] ]
  ]
  , If[
    ListQ[OptionValue[SortingFunction]],
    Table[Function[timesPair, f[timesPair[[1], timesPair[[2]], S,  $\Omega$ ]],
      {f, OptionValue[SortingFunction]}],
    Function[timesPair, OptionValue[SortingFunction][timesPair[[1], timesPair[[2]], S,  $\Omega$ ]]
  ]
]
]
End[];

```

Cutoff saddle finders

GetDoubleSaddlePoints

In[] :=

```

GetDoubleSaddlePoints::usage =
  "GetDoubleSaddlePoints[S,{tmin,tmax},{rmin,rmax},{ $\Omega$ min, $\Omega$ max}] finds a list of
  double solutions {t, $\tau$ , $\Omega$ } of the HHG temporal saddle-point equations, using a
  complex-valued  $\Omega$  in the range { $\Omega$ min, $\Omega$ max}, for action S, in the range {tmin, tmax}
  of recombination time and {rmin, rmax} of excursion time, where the ranges should
  indicate the lower-left and upper-right corners of rectangles in the complex plane.

  GetDoubleSaddlePoints[S,{{{tmin1,tmax1},{rmin1,rmax1},{ $\Omega$ min1, $\Omega$ max1}},{{tmin2,tmax2},{rmin2,rmax2},
  { $\Omega$ min2, $\Omega$ max2}},...]]

```

uses multiple variable ranges and combines the solutions.

`GetDoubleSaddlePoints[S, {{urange, vrange, Ω range}, ...}, IndependentVariables \rightarrow {u, v}]` uses the explicit independent temporal variables u and v to solve the equations and over the given ranges, where u and v can be any of `"RecombinationTime"`, `"IonizationTime"` and `"ExcursionTime"`, or their shorthands `"t"`, `"tt"` and `" τ "` resp.;

`GetDoubleSaddlePoints::error = "Errors encountered at tag `1`.";`

`ErrorReportingTag::usage = "";`

`Protect[ErrorReportingTag];`

`Begin["`Private`"];`

`Options[GetDoubleSaddlePoints] = Join[{SortingFunction \rightarrow ({#4 &, #2 &, #1 &}),
SelectionFunction \rightarrow (True &), ErrorReportingTag \rightarrow None,
IndependentVariables \rightarrow {"RecombinationTime", "ExcursionTime"}}, Options[FindComplexRoots]];`

`GetDoubleSaddlePoints[S_, {tmin_, tmax_},
{ τ min_, τ max_}, { Ω min_, Ω max_}, options : OptionsPattern[]] :=
GetDoubleSaddlePoints[S, {{tmin, tmax}, { τ min, τ max}, { Ω min, Ω max}}, options]`

`GetDoubleSaddlePoints[S_, variableRanges_, options : OptionsPattern[]] :=
Block[{equations, roots, t = Symbol["t"], tt = Symbol["tt"],
 τ = Symbol[" τ "], Ω , indVars, depVar, depVarRule, tolerances},
tolerances = SetTolerances[OptionValue[Tolerance], 3, OptionValue[WorkingPrecision]];
indVars = OptionValue[IndependentVariables] /.
{ "RecombinationTime" \rightarrow "t", "ExcursionTime" \rightarrow " τ ", "IonizationTime" \rightarrow "tt"};
depVar = First[DeleteCases[{"t", " τ ", "tt"}, Alternatives @@ indVars]];
depVarRule = depVar /. {"tt" \rightarrow {tt \rightarrow t - τ }, "t" \rightarrow {t \rightarrow tt + τ }, " τ " \rightarrow { τ \rightarrow t - tt}};
equations = {
D[S[t, tt], t] == Ω ,
D[S[t, tt], tt] == 0,
D[S[t, tt], {t, 2}] * D[S[t, tt], {tt, 2}] - D[S[t, tt], t, tt]2 == 0
} /. depVarRule;`

`SortBy[
DeleteDuplicates[
Flatten[Table[
Select[
Check[
roots = ({t, τ , Ω) /. depVarRule) /. (FindComplexRoots[
equations
, Evaluate[Sequence[
{Symbol[indVars[[1]], range[[1, 1]], range[[1, 2]]},`

```

        {Symbol[indVars[[2]], range[[2, 1]], range[[2, 2]],
         {Ω, range[[3, 1]], range[[3, 2]]}
      ]
    , Evaluate[Sequence @@ FilterRules[{options}, Options[FindComplexRoots]]]
    , SeedGenerator → RandomSobolComplexes
    , Seeds → 50
  ] /. {{t} → ({t, τ} /. depVarRule) → {}} (*to deal with empty results*)
,
If[OptionValue[ErrorReportingTag] ≠ None,
  Message[GetDoubleSaddlePoints::error, OptionValue[ErrorReportingTag]];
  roots
]
, Function[variableSet, OptionValue[SelectionFunction][variableSet[[1]], variableSet[[2]], S]]
]
, {range, variableRanges}], 1]
, Function[{variableSet1, variableSet2},
  And @@ Thread[Abs[variableSet1 - variableSet2] < tolerances] ]
]
, If[
  ListQ[OptionValue[SortingFunction]],
  Table[Function[variableSet, f[variableSet[[1]], variableSet[[2]], S, variableSet[[3]]],
    {f, OptionValue[SortingFunction]}],
  Function[variableSet, OptionValue[SortingFunction][variableSet[[1]],
    variableSet[[2]], S, variableSet[[3]]]
]
]
]
End[];

```

GetCutoffSaddlePoints

In[]:=

GetCutoffSaddlePoints::usage =

"GetCutoffSaddlePoints[S,{tmin,tmax},{rmin,rmax}] finds a list of solutions $\{t, \tau, \partial_t S, d_t^3 S\}$ of the HHG cutoff saddle-point equations $\{\partial_t S=0, d_t^2 S=0\}$ for action S, in the range {tmin, tmax} of recombination time and {rmin, rmax} of excursion time, where both ranges should be the lower-left and upper-right corners of rectangles in the complex plane.

GetCutoffSaddlePoints[S,{{{tmin₁,tmax₁},{rmin₁,rmax₁}},{{tmin₂,tmax₂},{rmin₂,rmax₂}},...] uses multiple time domains and combines the solutions.

GetCutoffSaddlePoints[S,{{urange,vrange},...},IndependentVariables→{u,v}] uses the explicit independent variables u and v to solve the equations and over the given ranges, where u and v can be any of "RecombinationTime", "IonizationTime"

```

and \"ExcursionTime\", or their shorthands \"t\", \"tt\" and \"\tau\" resp.";

GetCutoffSaddlePoints::error = "Errors encountered at tag `1`.";

Begin["`Private`"];
Options[GetCutoffSaddlePoints] =
  Join[{SortingFunction → (#2 &), SelectionFunction → (True &), ErrorReportingTag → None,
    IndependentVariables → {"RecombinationTime", "ExcursionTime"}}, Options[FindComplexRoots]];

GetCutoffSaddlePoints[S_, {tmin_, tmax_}, {rmin_, rmax_}, options : OptionsPattern[] :=
  GetCutoffSaddlePoints[S, {{tmin, tmax}, {rmin, rmax}}], options]

GetCutoffSaddlePoints[S_, timeRanges_, options : OptionsPattern[] :=
  Block[{equations, roots, t = Symbol["t"], tt = Symbol["tt"],
    \tau = Symbol["\tau"], indVars, depVar, depVarRule, tolerances, d1S, d3S},
    tolerances = SetTolerances[OptionValue[Tolerance], 2, OptionValue[WorkingPrecision]];
    indVars = OptionValue[IndependentVariables] /.
      {"RecombinationTime" → "t", "ExcursionTime" → "\tau", "IonizationTime" → "tt"};
    depVar = First[DeleteCases[{"t", "\tau", "tt"}, Alternatives @@ indVars]];
    depVarRule = depVar /. {"tt" → {tt → t - \tau}, "t" → {t → tt + \tau}, "\tau" → {\tau → t - tt}};
    equations = {
      D[S[t, tt], tt] == 0,
      D[S[t, tt], {t, 2}] * D[S[t, tt], {tt, 2}] - D[S[t, tt], t, tt]^2 == 0
    } /. depVarRule;
    d1S[t_, tt_] = ConstrainedDerivative[1][S][t, tt];
    d3S[t_, tt_] = ConstrainedDerivative[3][S][t, tt];

  Map[
    Association[Thread[{"t", "\tau", "S", "\partial_t S", "d_t^3 S"} → #]] &,
    SortBy[
      DeleteDuplicates[
        Flatten[Table[
          Select[
            Check[
              roots = ({t, \tau, S[t, t - \tau], d1S[t, t - \tau], d3S[t, t - \tau]} /. depVarRule) /. (FindComplexRoots[
                equations
                , Evaluate[Sequence[{Symbol[indVars[[1]], range[[1, 1], range[[1, 2]],
                  {Symbol[indVars[[2]], range[[2, 1], range[[2, 2]]}]]
                , Evaluate[Sequence @@ FilterRules[{options}, Options[FindComplexRoots]]]
                , SeedGenerator → RandomSobolComplexes
                , Seeds → 50
              ] /. {{}} → ({t, \tau} /. depVarRule) → {}})] (*to deal with empty results*)
            ,
          If[OptionValue[ErrorReportingTag] ≠ None,
            Message[GetCutoffSaddlePoints::error, OptionValue[ErrorReportingTag]]];

```

```

        roots
    ]
    , Function[timesPair, OptionValue[SelectionFunction][timesPair[[1], timesPair[[2]], S]]
    ]
    , {range, timeRanges}], 1]
    , Function[{timesPair1, timesPair2},
        And @@ Thread[Abs[timesPair1 - timesPair2][[1, 2]] < tolerances] ]
    ]
    , If[
        ListQ[OptionValue[SortingFunction]],
        Table[
            Function[timesPair, f[timesPair[[1], timesPair[[2]], S]], {f, OptionValue[SortingFunction]}],
            Function[timesPair, OptionValue[SortingFunction][timesPair[[1], timesPair[[2]], S]]
        ]
    ]
    ]
    ]
    ]
End[];

```

ClassifyQuantumOrbits

In[]:=

```

ClassifyQuantumOrbits::usage =
  "ClassifyQuantumOrbits[saddlePoints,f] sorts an indexed set of saddle
  points of the form  $\langle |\Omega_1 \rightarrow \{\{t_{11}, \tau_{11}\}, \{t_{12}, \tau_{12}\}, \dots\} \dots \rangle$  using a function f,
  which should turn  $f[t, \tau, \Omega]$  into an appropriate label, and returns an
  association of the form  $\langle | \text{label}_1 \rightarrow \langle |\Omega_1 \rightarrow \langle | 1 \rightarrow \{t, \tau\}, 2 \rightarrow \{t, \tau\}, \dots \rangle, \dots \rangle, \dots \rangle$ .

ClassifyQuantumOrbits[saddlePoints,f,sortFunction] uses the function sortFunction to sort
  the sets of saddle points  $\{\{t_{11}, \tau_{11}\}, \{t_{12}, \tau_{12}\}, \dots\}$  for each label and harmonic energy.

ClassifyQuantumOrbits[saddlePoints,f,sortFunction,DiscardedLabels $\rightarrow$ {label1,label2,...}]
  specifies a list of labels to discard from the final output.";
DiscardedLabels::usage = "DiscardedLabels is an option for ClassifyQuantumOrbits
  which specifies a list of labels to discard from the final output.";

Begin["`Private`"];

Options[ClassifyQuantumOrbits] = {DiscardedLabels  $\rightarrow$  {}};
Protect[DiscardedLabels];

ClassifyQuantumOrbits[saddlePointList_,
  classifierFunction_, sortingFunction_:Sort, OptionsPattern[]] := Map[
  Composition[
    Association,
    MapIndexed[ $\#2[[1]] \rightarrow \#1 \&$ ],
    sortingFunction
  ],
  Delete[
    AssociationTranspose[
      MapIndexed[
        GroupBy[classifierFunction @@  $\# \&$ ][
          Flatten /@ Transpose[{ $\#1$ , ConstantArray[ $\#2[[1]]$ , 1], Length[ $\#1$ ]}]] &
        , saddlePointList][All, All, All, {1, 2}]
      ]
    , List /@ OptionValue[DiscardedLabels]
    , {2}]
End[];

```

ReperiodSaddles

```

In[ ]:= ClearAll[ReperiodSaddles]
ReperiodSaddles::usage =
  "ReperiodSaddles[{{t1,τ1},{t2,τ2},...},f] readjusts the assigned cycle of
  the saddle points {ti,τi}, returning the list {{t1+f[t1,τ1],τ1},...}.

ReperiodSaddles[<|Ω1→{{t11,τ11},...},Ω2→...|>,f]
  reperiods saddle-point pairs in a harmonic-energy-indexed association.

ReperiodSaddles[<|label1→<|Ω1→{{t11,τ11},...},...|>,...|>,f]
  reperiods saddle-point pairs of a classified set of saddle points.";

Begin["`Private`"];

ReperiodSaddles[pair_ /; Depth[pair] == 2, f_] := {pair[[1]] + f[pair[[1]], pair[[2]]], pair[[2]]}
ReperiodSaddles[association_, f_] := Apply[f, association, {Depth[association] - 2}]

End[];

```

HessianRoot

```

In[ ]:= HessianRoot::usage = "HessianRoot[S,t,τ] calculates the Hessian root  $\sqrt{\frac{(2\pi)^2}{i^2 \text{Det}[\partial_{\{t, \tau\}}^2 S]}}$ .";

Begin["`Private`"];

HessianRoot[S_, t_, τ_] :=  $\sqrt{\frac{2\pi}{i \text{Derivative}[0, 2][S][t, t - \tau]}}$ 

 $\sqrt{\frac{2\pi \text{Derivative}[0, 2][S][t, t - \tau]}{i (\text{Derivative}[2, 0][S][t, t - \tau] \times \text{Derivative}[0, 2][S][t, t - \tau] - \text{Derivative}[1, 1][S][t, t - \tau]^2)}}$ 

End[];

```

FindStokesTransitions

```

In[ ]:= FindStokesTransitions::usage =
  "FindStokesTransitions[S,<|Ω1→<|1→{t11,τ11},2→{t12,τ12}|>,Ω2→<|1→{t21,τ21},2→{t22,τ22}|>,...
  |>] finds the set {{ΩS},{ΩAS},n} of the Stokes and anti-Stokes transition
  energies for the given set of saddle points, where Re(S) changes sign after the
  ΩS and Im(S) changes sign after the ΩAS, and n is the index of the member of
  the pair that should be chosen after the transition (taken as the member with

```


a positive imaginary part of the action at the largest Ω_i in the given keys).

```
FindStokesTransitions[S, <|label1→<|Ω1→...>|>] finds the Stokes transitions for the given
  set of saddle-point curve pairs, and returns them labeled with the labeli.";
FindStokesTransitions::saddleno = "FindStokesTransitions called with `1`
  of `2` saddle-point sets of length different from 2, with set
  length structure `3`. Excluding those sets from the calculation.";
FindStokesTransitions::multipleS = "FindStokesTransitions found multiple
  Stokes transitions; using `1` to return a single transition.";
FindStokesTransitions::multipleAS = "FindStokesTransitions found multiple
  anti-Stokes transitions; using `1` to return a single transition.";
ChooserFunction::usage = "ChooserFunction is an option for FindStokesTransitions
  that specifies which transition to take if there are multiple transitions
  in the given dataset. The default is Last and gives the one with higher
  energy; to get the full set of transitions found use Full or Identity.";
ReperiodingFunction::usage = "ReperiodingFunction is an option for FindStokesTransitions,
  SPAdipole and UAdipole which specifies a function f[t,τ] of recombination
  time t and excursion time τ that will be used to re-period the pairs
  {t,τ} into the form {t+f[t,τ],τ}. The default is Function[0], but
  if pairs are split it can be useful to set ReperiodingFunction to
      Function[{t,τ}, Floor[-Re[t-τ],  $\frac{2\pi}{\omega}$ ]] for ω the carrier frequency. In general,
  however, it is preferable to do this in a single go using ReperiodSaddles.";

Begin["`Private`"];

Protect[ReperiodingFunction, ChooserFunction];
Options[FindStokesTransitions] =
  {ReperiodingFunction → Function[{t, τ}, 0], ChooserFunction → Automatic};

FindStokesTransitions[S_,
  deeperAssociation_ /; Depth[deeperAssociation] == 5, options : OptionsPattern[] := Map[
    FindStokesTransitions[S, #, options] &,
    deeperAssociation
  ]

FindStokesTransitions[S_, saddlesAssociation_, options : OptionsPattern[] :=
  Block[{reducedSaddlesAssociation, actionList, signsList, s, processor},
    reducedSaddlesAssociation = KeySort[Select[saddlesAssociation, Length[#] == 2 &]];
    If[Length[saddlesAssociation] - Length[reducedSaddlesAssociation] > 0,
      Message[FindStokesTransitions::saddleno,
        Length[saddlesAssociation] - Length[reducedSaddlesAssociation],
        Length[saddlesAssociation], First /@ Tally /@ Split[Values[Length /@ saddlesAssociation]]
      ]
    ];
```

```

actionList = ReIm[
  Map[(*reduces each  $\Omega \rightarrow \langle |1 \rightarrow S_1, 2 \rightarrow S_2 \rangle$  to  $\Omega \rightarrow (S_1 - S_2)$ *)
    Apply[Subtract],
    MapIndexed[(*reduces each  $\Omega \rightarrow \langle |1 \rightarrow \{t_1, \tau_1\}, 2 \rightarrow \{t_2, \tau_2\} \rangle$  to  $\Omega \rightarrow \langle |1 \rightarrow S_1, 2 \rightarrow S_2 \rangle$ *)
      With[{t = #1[[1]] + OptionValue[ReperiodingFunction][#1[[1]], #2[[2]]],  $\tau$  = #1[[2]],  $\Omega$  = #2[[1, 1]]},
        S[t, t -  $\tau$ ] -  $\Omega$  t
      ] &
    , reducedSaddlesAssociation, {2}
  ]
];
signsList = Sign[Times[
  Rest[actionList],
  AssociationThread[Rest[Keys[actionList]], Most[Values[actionList]]]
]];
processor = OptionValue[ChooserFunction] /. {Automatic -> Last, Full -> Identity};
If[Length[Keys[Select[signsList, #1 < 0 &]]] > 1,
  Message[FindStokesTransitions::multipleS, processor]];
If[Length[Keys[Select[signsList, #2 < 0 &]]] > 1,
  Message[FindStokesTransitions::multipleAS, processor]];
{
  processor[Keys[Select[signsList, #1 < 0 &]] /. {} -> {Missing["No transition"]}],
  processor[Keys[Select[signsList, #2 < 0 &]] /. {} -> {Missing["No transition"]}],
  Sign[Last[actionList][[2]] /. {1 -> 2, -1 -> 1}
}
]

End[];

```

SPAdipole

In[]:=

```

SPAdipole::usage =
  "SPAdipole[S,prefactor, $\Omega$ ,{t, $\tau$ }] returns the saddle-point approximation amplitude
  corresponding to action  $S[t,t-\tau]-\Omega t$  and the given prefactor[t,t- $\tau$ ].

  SPAdipole[S,prefactor, $\Omega$ ,<|1→{t1, $\tau$ 1},2→{t2, $\tau$ 2},...|>] returns the total harmonic-dipole
  contribution in the saddle-point approximation from the specified saddle points.

  SPAdipole[S,prefactor, $\Omega$ ,<|1→{t1, $\tau$ 1},2→{t2, $\tau$ 2}|>,transition] uses the given Stokes
  transition set to drop the relevant saddle after the anti-Stokes transition.";
SPAdipole::wrongno = "SPAdipole called with a Stokes transition but with an input association
  of length `1` at harmonic energy  $\Omega$ =`2`. Reverting to unstructured evaluation.";
SPAdipole::invldtrns = "SPAdipole called with invalid Stokes transition
  set `1`. Reverting to unstructured evaluation.";

Begin["`Private`"];

Options[SPAdipole] = {ReperiodingFunction → Function[{t,  $\tau$ ], 0}};

SPAdipole[S_, prefactor_,  $\Omega$ _, {t_,  $\tau$ _}, options : OptionsPattern[]] :=
  Block[{tr = t + OptionValue[ReperiodingFunction][t,  $\tau$ ]},
    HessianRoot[S, tr,  $\tau$ ] × prefactor[tr, tr -  $\tau$ ] Exp[- $i$  S[tr, tr -  $\tau$ ] +  $i$   $\Omega$  tr]
  ]
SPAdipole[S_, prefactor_,  $\Omega$ _, times_Association, options : OptionsPattern[]] := Block[{},
  Total[SPAdipole[S, prefactor,  $\Omega$ , #, options] & /@ times]
]

SPAdipole[S_, prefactor_,  $\Omega$ _, times_Association,
  transition_, options : OptionsPattern[]] := Block[{},
  If[! NumberQ[transition[[2]]], Message[SPAdipole::invldtrns, transition];
    Return[SPAdipole[S, prefactor,  $\Omega$ , times]]];
  If[Length[times] ≠ 2, Message[SPAdipole::wrongno, Length[times],  $\Omega$ ];
    Return[SPAdipole[S, prefactor,  $\Omega$ , times, options]]];
  If[ $\Omega$  < transition[[1]],
    SPAdipole[S, prefactor,  $\Omega$ , times, options],
    SPAdipole[S, prefactor,  $\Omega$ , KeySelect[times, # == transition[[3]] &], options]
  ]
]

End[];

```

UAdipole

In[]:=

```

UAdipole::usage =
  "UAdipole[S,prefactor, $\Omega$ ,<|1→{t1, $\tau$ 1},2→{t2, $\tau$ 2},...|>,transition] returns the total

```

```

    harmonic-dipole contribution in the uniform approximation from the
    specified saddle points, using the action  $S[t, t-\tau] - \Omega t$  and prefactor  $[t, t-\tau]$ ,
    and taking the given Stokes transition set as a reference.";
UAdipole::saddleno = "UAdipole called with `1` time pairs at  $\Omega = `2`$ .
    Reverting to the saddle-point approximation for this set.";
UAdipole::invldtrns = "UAdipole called with invalid Stokes transition set
    `1`. Reverting to the saddle-point approximation for this set.";

Begin["`Private`"];

Options[UAdipole] = {ReperiodingFunction → Function[{t,  $\tau$ }, 0]};

UAdipole[S_, prefactor_,  $\Omega$ _, times_, transition_, options: OptionsPattern[]]:= (
    If[Length[times] ≠ 2, Message[UAdipole::saddleno, Length[times],  $\Omega$ ];
    Return[SPAdipole[S, prefactor,  $\Omega$ , times]]];
    If[! NumberQ[transition[[2]]], Message[UAdipole::invldtrns, transition];
    Return[SPAdipole[S, prefactor,  $\Omega$ , times]]];
    Block[
        {A1, A2, S1, S2, Ss, Sm, z,
            t1 = times[1][1] + OptionValue[ReperiodingFunction][times[1][1], times[1][2]], r1 = times[1][2],
            t2 = times[2][1] + OptionValue[ReperiodingFunction][times[2][1], times[2][2]], r2 = times[2][2]},
        A1 = HessianRoot[S, t1, r1] × prefactor[t1, t1 - r1];
        S1 = S[t1, t1 - r1] -  $\Omega$  t1;
        A2 = HessianRoot[S, t2, r2] × prefactor[t2, t2 - r2];
        S2 = S[t2, t2 - r2] -  $\Omega$  t2;
        Ss =  $\frac{S1 + S2}{2}$ ; Sm =  $\frac{S1 - S2}{2}$ ;
        If[ $\Omega < \text{transition}[[2]]$ ,  $z = \left(-\frac{3}{2} \text{Sm}\right)^{2/3}$ ,  $z = \left(-\frac{3}{2} \text{Sm}\right)^{2/3} \text{Exp}\left[i \left(\text{transition}[[3]] /. \{2 \rightarrow -1, 1 \rightarrow 1\}\right) \frac{2\pi}{3}\right]$ ];
         $\sqrt{6\pi \text{Sm}} \text{Exp}\left[-i \text{Ss} + i \frac{\pi}{4}\right] \left( \frac{A1 - i A2}{2} \frac{\text{AiryAi}[-z]}{\sqrt{z}} + i \frac{A1 + i A2}{2} \frac{\text{AiryAi}'[-z]}{z} \right)$ 
    ]
)

End[];

```

Package closure

End of package

```
EndPackage[];
```

Add to distributed contexts.

In[]:=

```
DistributeDefinitions["RBSFA`"];
```