Tutorial-Method_of_Lines-C_Code_Generation

May 12, 2023

-1 Generating C Code to implement Method of Lines Timestepping for Explicit Runge Kutta Methods

-1.1 Authors: Zach Etienne & Brandon Clark

-1.2 This tutorial notebook generates three blocks of C Code in order to perform Method of Lines timestepping.

Notebook Status: Validated

Validation Notes: This tutorial notebook has been confirmed to be self-consistent with its corresponding NRPy+ module, as documented below. All Runge-Kutta (RK) Butcher tables were validated using truncated Taylor series in a separate module. Finally, C-code implementation of RK4 was validated against a trusted version. C-code implementations of other RK methods seem to work as expected in the context of solving the scalar wave equation in Cartesian coordinates.

-1.2.1 NRPy+ Source Code for this module:

- MoLtimestepping/C_Code_Generation.py
- MoLtimestepping/RK_Butcher_Table_Dictionary.py (**Tutorial**) stores the Butcher tables for the explicit Runge Kutta methods.

-1.3 Introduction:

When numerically solving a partial differential equation initial-value problem, subject to suitable boundary conditions, we implement Method of Lines to "integrate" the solution forward in time.

-1.3.1 The Method of Lines:

Once we have the initial data for a PDE, we "evolve it forward in time", using the Method of Lines. In short, the Method of Lines enables us to handle 1. the **spatial derivatives** of an initial value problem PDE using **standard finite difference approaches**, and 2. the **temporal derivatives** of an initial value problem PDE using **standard strategies for solving ordinary differential equations (ODEs), like Runge Kutta methods** so long as the initial value problem PDE can be written in the first-order-in-time form

$$\partial_t \vec{f} = \mathbf{M}\ \vec{f},$$

where $\mathbf{M}$ is an $N \times N$ matrix containing only *spatial* differential operators that act on the N-element column vector $\vec{f}$. $\mathbf{M}$ may not contain t or time derivatives explicitly; only *spatial* partial derivatives are allowed to appear inside $\mathbf{M}$.

You may find the module Tutorial-ScalarWave extremely helpful as an example for implementing the Method of Lines for solving the Scalar Wave equation in Cartesian coordinates.

-1.3.2 Generating the C code:

This module describes how core C functions are generated to implement Method of Lines timestepping for a specified RK method. There are three core functions:

1. allocate memory for gridfunctions,
2. step forward the solution one full timestep,
3. and free memory for gridfunctions.

The first function is called first, then the second function is repeated within a loop to a fixed "final" time (such that the end state of each iteration is the initial state for the next iteration), and the third function is called at the end of the calculation.

The generated codes are essential for a number of Start-to-Finish example tutorial notebooks that demonstrate how to numerically solve hyperbolic PDEs.

0 Table of Contents

This notebook is organized as follows

1. Step 1: Initialize needed Python/NRPy+ modules
2. Step 2: Checking if Butcher Table is Diagonal
3. Step 3: Generating the C Code
    1. Step 3.a: `generate_gridfunction_names()`: Uniquely and descriptively assign names to sets of gridfunctions
    2. Step 3.b: Memory allocation: `MoL_malloc_y_n_gfs()` and `MoL_malloc_non_y_n_gfs()`
    3. Step 3.c: Take one Method of Lines time step: `MoL_step_forward_in_time()`
    4. Step 3.d: Memory deallocation: `MoL_free_memory()`
    5. Step 3.e: Define & register `MoL_gridfunctions_struct` in `NRPy_basic_defines.h`: `NRPy_basic_defines_MoL_timestepping_struct()`
    6. Step 3.f: Add all MoL C codes to C function dictionary, and add MoL definitions to `NRPy_basic_defines.h`: `register_C_functions_and_NRPy_basic_defines()`
4. Step 4: Code Validation against `MoLtimestepping.RK_Butcher_Table_Generating_C_Code` NRPy+ module
5. Step 5: Output this notebook to LaTeX-formatted PDF file

1 Step 1: Initialize needed Python/NRPy+ modules [Back to top]

Let's start by importing all the needed modules from Python/NRPy+:

[1]:
```
import sympy as sp   # Import SymPy, a computer algebra system written entirely in Python
import os, sys       # Standard Python modules for multiplatform OS-level functions
from MoLtimestepping.RK_Butcher_Table_Dictionary import Butcher_dict
from outputC import add_to_Cfunction_dict, indent_Ccode, outC_NRPy_basic_defines_h_dict, superfast_uniq, outputC   # NRPy+: Basic C code␣
 ↪output functionality
```

2 Step 2: Checking if a Butcher table is Diagonal [Back to top]

A diagonal Butcher table takes the form

$$\begin{array}{c|cccccc} 0 & & & & & & \\ a_1 & a_1 & & & & & \\ a_2 & 0 & a_2 & & & & \\ a_3 & 0 & 0 & a_3 & & & \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \\ a_s & 0 & 0 & 0 & \cdots & a_s & \\ \hline & b_1 & b_2 & b_3 & \cdots & b_{s-1} & b_s \end{array},$$

where s is the number of required predictor-corrector steps for a given RK method (see Butcher, John C. (2008)). One known diagonal RK method is the classic RK4 represented in Butcher table form below.

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

Diagonal Butcher tables are nice when it comes to saving required memory space. Each new step for a diagonal RK method, when computing the new k_i, does not depend on the previous calculation, and so there are ways to save memory. Significantly so in large three-dimensional spatial grid spaces.

[2]:
```python
# Check if Butcher Table is diagonal
def diagonal(key):
    # Get the Butcher table corresponding to the provided key
    Butcher = Butcher_dict[key][0]

    # Establish the number of rows to check for diagonal trait, all but the last row
    L = len(Butcher) - 1

    # Initialize the Butcher table row index
    row_idx = 0

    # Check all the desired rows
    for i in range(L):
        # Check each element before the diagonal element in a row
        for j in range(1, row_idx):
            # If any non-diagonal coefficient is non-zero, then the table is not diagonal
            if Butcher[i][j] != sp.sympify(0):
                return False
        # Update to check the next row
        row_idx += 1

    # If all non-diagonal elements are zero, the table is diagonal
    return True
```

3 Step 3: Generating the C Code [Back to top]

The following sections build up the C code for implementing the Method of Lines timestepping algorithm for solving hyperbolic PDEs.

First, an important note on efficiency.

Memory efficiency is incredibly important here, as $\vec{f}$ is usually the largest object in memory.

If we only made use of the Butcher tables without concern for memory efficiency, `generate_gridfunction_names()` and `MoL_step_forward_in_time()` would be very simple functions.

It turns out that several of the Runge-Kutta-like methods in MoL can be made more efficient; for example "RK4" can be performed using only 4 "timelevels" of $\vec{f}$ in memory (i.e., a total memory usage of `sizeof(f) * 4`). A naive implementation might use 5 or 6 copies. RK-like methods that have diagonal Butcher tables can be made far more efficient than the naive approach.

Exercise to student: Improve the efficiency of other RK-like methods.

3.1 Step 3.a: `generate_gridfunction_names()`: Uniquely and descriptively assign names to sets of gridfunctions [Back to top]

`generate_gridfunction_names()` names gridfunctions to be consistent with a given RK substep. For example, we might call the set of gridfunctions stored at substep k_1 `k1_gfs`.

[3]:
```
def generate_gridfunction_names(MoL_method="RK4"):
    """
    Generate gridfunction names for the specified Method of Lines (MoL) method.

    :param MoL_method: The MoL method to generate gridfunction names for.
    :return: A tuple containing y_n_gridfunctions, non_y_n_gridfunctions_list,
             diagnostic_gridfunctions_point_to, and diagnostic_gridfunctions2_point_to.
    """
    # y_n_gridfunctions store data for the vector of gridfunctions y_i at t_n,
    # the start of each MoL timestep.
    y_n_gridfunctions = "y_n_gfs"
    # non_y_n_gridfunctions are needed to compute the data at t_{n+1}.
    # Often labeled with k_i in the name, these gridfunctions are *not*
    # needed at the start of each timestep.
    non_y_n_gridfunctions_list = []

    # Diagnostic output gridfunctions diagnostic_output_gfs & diagnostic_output_gfs2.
    diagnostic_gridfunctions2_point_to = ""

    if diagonal(MoL_method) and "RK3" in MoL_method:
        non_y_n_gridfunctions_list.append("k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs")
        non_y_n_gridfunctions_list.append("k2_or_y_nplus_a32_k2_gfs")
        diagnostic_gridfunctions_point_to = "k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs"
        diagnostic_gridfunctions2_point_to = "k2_or_y_nplus_a32_k2_gfs"
    else:
        if not diagonal(MoL_method):  # Allocate memory for non-diagonal Butcher tables
            # Determine the number of k_i steps based on length of Butcher Table
            num_k = len(Butcher_dict[MoL_method][0]) - 1
            # For non-diagonal tables, an intermediate gridfunction "next_y_input" is used for rhs evaluations
            non_y_n_gridfunctions_list.append("next_y_input_gfs")
            for i in range(num_k):  # Need to allocate all k_i steps for a given method
                non_y_n_gridfunctions_list.append("k{}_gfs".format(i + 1))
            diagnostic_gridfunctions_point_to = "k1_gfs"
            if "k2_gfs" in non_y_n_gridfunctions_list:
                diagnostic_gridfunctions2_point_to = "k2_gfs"
            else:
                print("MoL WARNING: No gridfunction group available for diagnostic_output_gfs2")
        else:  # Allocate memory for diagonal Butcher tables, which use a "y_nplus1_running_total gridfunction"
            non_y_n_gridfunctions_list.append("y_nplus1_running_total_gfs")
            if MoL_method != 'Euler':  # Allocate memory for diagonal Butcher tables that aren't Euler
                # Need k_odd for k_1,3,5... and k_even for k_2,4,6...
                non_y_n_gridfunctions_list.append("k_odd_gfs")
                non_y_n_gridfunctions_list.append("k_even_gfs")
            diagnostic_gridfunctions_point_to = "y_nplus1_running_total_gfs"
            diagnostic_gridfunctions2_point_to = "k_odd_gfs"
    non_y_n_gridfunctions_list.append("auxevol_gfs")

    return y_n_gridfunctions, non_y_n_gridfunctions_list, \
        diagnostic_gridfunctions_point_to, diagnostic_gridfunctions2_point_to
```

3.2 Step 3.b: Memory allocation: `MoL_malloc_y_n_gfs()` and `MoL_malloc_non_y_n_gfs()`: [Back to top]

Generation of C functions `MoL_malloc_y_n_gfs()` and `MoL_malloc_non_y_n_gfs()` read the full list of needed lists of gridfunctions, provided by (Python) function `generate_gridfunction_names()`, to allocate space for all gridfunctions.

[4]:
```
# add_to_Cfunction_dict_MoL_malloc() registers
#           MoL_malloc_y_n_gfs() and
#           MoL_malloc_non_y_n_gfs(), which allocate memory for
#           the indicated sets of gridfunctions
def add_to_Cfunction_dict_MoL_malloc(MoL_method, which_gfs):
    includes = ["NRPy_basic_defines.h", "NRPy_function_prototypes.h"]
    # Create a description for the function
    desc = "Method of Lines (MoL) for \"" + MoL_method + "\" method: Allocate memory for \"" + which_gfs + "\" gridfunctions\n"
    desc += "   * y_n_gfs are used to store data for the vector of gridfunctions y_i at t_n, at the start of each MoL timestep\n"
    desc += "   * non_y_n_gfs are needed for intermediate (e.g., k_i) storage in chosen MoL method\n"
    c_type = "void"

    y_n_gridfunctions, non_y_n_gridfunctions_list, diagnostic_gridfunctions_point_to, diagnostic_gridfunctions2_point_to = \
        generate_gridfunction_names(MoL_method=MoL_method)

    # Determine which gridfunctions to allocate memory for
    if which_gfs == "y_n_gfs":
        gridfunctions_list = [y_n_gridfunctions]
    elif which_gfs == "non_y_n_gfs":
        gridfunctions_list = non_y_n_gridfunctions_list
    else:
        print("ERROR: which_gfs = \"" + which_gfs + "\" unrecognized.")
        sys.exit(1)

    name = "MoL_malloc_" + which_gfs
    params = "const paramstruct *restrict params, MoL_gridfunctions_struct *restrict gridfuncs"

    # Generate the body of the function
    body = "const int Nxx_plus_2NGHOSTS_tot = Nxx_plus_2NGHOSTS0*Nxx_plus_2NGHOSTS1*Nxx_plus_2NGHOSTS2;\n"
    for gridfunctions in gridfunctions_list:
        num_gfs = "NUM_EVOL_GFS"
        if gridfunctions == "auxevol_gfs":
            num_gfs = "NUM_AUXEVOL_GFS"
        body += "gridfuncs->" + gridfunctions + " = (REAL *restrict)malloc(sizeof(REAL) * " + num_gfs + " * Nxx_plus_2NGHOSTS_tot);\n"
    body += "\ngridfuncs->diagnostic_output_gfs  = gridfuncs->" + diagnostic_gridfunctions_point_to + ";\n"
    body += "\ngridfuncs->diagnostic_output_gfs2 = gridfuncs->" + diagnostic_gridfunctions2_point_to + ";\n"

    # Add the function to the C function dictionary
    add_to_Cfunction_dict(
        includes=includes,
        desc=desc,
        c_type=c_type, name=name, params=params,
        body=indent_Ccode(body, "  "),
        rel_path_to_Cparams=os.path.join("."))
```

3.3 Step 3.c: Take one Method of Lines time step: `MoL_step_forward_in_time()` [Back to top]

An MoL step consists in general of a series of Runge-Kutta-like substeps, and the `MoL_step_forward_in_time()` C function pulls together all of these substeps.

The basic C code for an MoL substep, set up by the Python function `single_RK_substep()` below, is as follows.

1. Evaluate the right-hand side of $\partial_t \vec{f} = \texttt{RHS}$, to get the time derivative of the set of gridfunctions $\vec{f}$ at our current time.
2. Perform the Runge-Kutta update, which depends on $\partial_t \vec{f}$ on the current and sometimes previous times.
3. Call post-right-hand side functions as desired.

The `single_RK_substep_input_symbolic()` function generates the C code for performing the above steps, applying substitutions for e.g., `RK_INPUT_GFS` and `RK_OUTPUT_GFS` as appropriate. `single_RK_substep_input_symbolic()` supports SIMD-capable code generation.

[5]:
```
# single_RK_substep_input_symbolic() performs necessary replacements to
#   define C code for a single RK substep
#   (e.g., computing k_1 and then updating the outer boundaries)
def single_RK_substep_input_symbolic(comment_block, substep_time_offset_dt, RHS_str, RHS_input_str, RHS_output_str, RK_lhs_list,␣
↪RK_rhs_list,
                                     post_RHS_list, post_RHS_output_list, enable_SIMD=False,
                                     gf_aliases="", post_post_RHS_string=""):
    return_str = comment_block + "\n"
    substep_time_offset_str = "{:.17e}".format(float(substep_time_offset_dt))
    return_str += "griddata->params.time = time_start + " + substep_time_offset_str + " * griddata->params.dt;\n"

    # Ensure all input lists are lists
    RK_lhs_list = [RK_lhs_list] if not isinstance(RK_lhs_list, list) else RK_lhs_list
    RK_rhs_list = [RK_rhs_list] if not isinstance(RK_rhs_list, list) else RK_rhs_list
    post_RHS_list = [post_RHS_list] if not isinstance(post_RHS_list, list) else post_RHS_list
    post_RHS_output_list = [post_RHS_output_list] if not isinstance(post_RHS_output_list, list) else post_RHS_output_list

    return_str += "{\n" + indent_Ccode(gf_aliases, "  ")
    indent = "  "

    # Part 1: RHS evaluation
    return_str += indent_Ccode(str(RHS_str).replace("RK_INPUT_GFS",  str(RHS_input_str).replace("gfsL", "gfs")).
                              replace("RK_OUTPUT_GFS", str(RHS_output_str).replace("gfsL", "gfs")) + "\n", indent=indent)

    # Part 2: RK update
    if enable_SIMD:
        return_str += "#pragma omp parallel for\n"
        return_str += indent + "for(int i=0;i<Nxx_plus_2NGHOSTS0*Nxx_plus_2NGHOSTS1*Nxx_plus_2NGHOSTS2*NUM_EVOL_GFS;i+=SIMD_width) {\n"
    else:
        return_str += indent + "LOOP_ALL_GFS_GPS(i) {\n"

    var_type = "REAL_SIMD_ARRAY" if enable_SIMD else "REAL"
    RK_lhs_str_list = [indent + "const REAL_SIMD_ARRAY __RHS_exp_" + str(i) if enable_SIMD else indent + str(el).replace("gfsL",␣
↪"gfs[i]") for i, el in enumerate(RK_lhs_list)]

    read_list = [read for el in RK_rhs_list for read in list(sp.ordered(el.free_symbols))]
    read_list_unique = superfast_uniq(read_list)

    for el in read_list_unique:
        if str(el) != "params->dt":
            if enable_SIMD:
                simd_el = str(el).replace("gfsL", "gfs[i]")
                return_str += "{}  const {} {} = ReadSIMD(&{});\n".format(indent, var_type, str(el), simd_el)
            else:
                return_str += "{}  const {} {} = {};\n".format(indent, var_type, str(el),
                                                               str(el).replace("gfsL", "gfs[i]"))

    if enable_SIMD:
        return_str += "{}  const REAL_SIMD_ARRAY DT = ConstSIMD(params->dt);\n".format(indent)

    pre_indent = "2"
    kernel = outputC(RK_rhs_list, RK_lhs_str_list, filename="returnstring",
                     params="includebraces=False,preindent="+pre_indent+",outCverbose=False,enable_SIMD="+str(enable_SIMD))
    if enable_SIMD:
        return_str += kernel.replace("params->dt", "DT")
        for i, el in enumerate(RK_lhs_list):
            return_str += "  WriteSIMD(&" + str(el).replace("gfsL", "gfs[i]") + ", __RHS_exp_" + str(i) + ");\n"
    else:
        return_str += kernel

    return_str += indent + "}\n"

    # Part 3: Call post-RHS functions
    for post_RHS, post_RHS_output in zip(post_RHS_list, post_RHS_output_list):
        return_str += indent_Ccode(post_RHS.replace("RK_OUTPUT_GFS", str(post_RHS_output).replace("gfsL", "gfs")))

    return_str += "}\n"

    for post_RHS, post_RHS_output in zip(post_RHS_list, post_RHS_output_list):
        return_str += indent_Ccode(post_post_RHS_string.replace("RK_OUTPUT_GFS", str(post_RHS_output).replace("gfsL", "gfs")), "")

    return return_str
```

In the `add_to_Cfunction_dict_MoL_step_forward_in_time()` Python function below, we construct and register the core C function for MoL timestepping: `MoL_step_forward_in_time()`. `MoL_step_forward_in_time()` implements Butcher tables for Runge-Kutta-like methods, leveraging the `single_RK_substep()` helper function above as needed. Again, we aim for maximum memory efficiency so that, e.g., RK4 needs to store only 4 levels of $\vec{f}$.

[6]:
```
def add_to_Cfunction_dict_MoL_step_forward_in_time(MoL_method,
                                                    RHS_string = "", post_RHS_string = "", post_post_RHS_string="",
                                                    enable_rfm=False, enable_curviBCs=False, enable_SIMD=False):
    includes = ["NRPy_basic_defines.h", "NRPy_function_prototypes.h"]
    if enable_SIMD:
        includes += [os.path.join("SIMD", "SIMD_intrinsics.h")]
    desc  = "Method of Lines (MoL) for \"" + MoL_method + "\" method: Step forward one full timestep.\n"
    c_type = "void"
    name = "MoL_step_forward_in_time"
    params = "griddata_struct *restrict griddata"

    indent = ""  # We don't bother with an indent here.

    body = indent + "// C code implementation of -={ " + MoL_method + " }=- Method of Lines timestepping.\n\n"
    body += "// First set the initial time:\n"
    body += "const REAL time_start = griddata->params.time;\n"

    y_n_gridfunctions, non_y_n_gridfunctions_list, _throwaway, _throwaway2 = generate_gridfunction_names(MoL_method)

    gf_prefix = "griddata->gridfuncs."

    gf_aliases = """// Set gridfunction aliases from gridfuncs struct
REAL *restrict """ + y_n_gridfunctions + " = "+gf_prefix + y_n_gridfunctions + """;  // y_n gridfunctions
// Temporary timelevel & AUXEVOL gridfunctions:\n"""
    for gf in non_y_n_gridfunctions_list:
        gf_aliases += "REAL *restrict " + gf + " = "+gf_prefix + gf + ";\n"

    gf_aliases += "paramstruct *restrict params = &griddata->params;\n"
    if enable_rfm:
        gf_aliases += "const rfm_struct *restrict rfmstruct = &griddata->rfmstruct;\n"
    else:
        gf_aliases += "REAL *restrict xx[3]; for(int ww=0;ww<3;ww++) xx[ww] = griddata->xx[ww];\n"
    if enable_curviBCs:
        gf_aliases += "const bc_struct *restrict bcstruct = &griddata->bcstruct;\n"
    for i in ["0", "1", "2"]:
        gf_aliases += "const int Nxx_plus_2NGHOSTS" + i + " = griddata->params.Nxx_plus_2NGHOSTS" + i + ";\n"

    # Implement Method of Lines (MoL) Timestepping
    Butcher = Butcher_dict[MoL_method][0]   # Get the desired Butcher table from the dictionary
    num_steps = len(Butcher)-1  # Specify the number of required steps to update solution

    dt = sp.Symbol("params->dt", real=True)

    if diagonal(MoL_method) and "RK3" in MoL_method:
        # Diagonal RK3 only!!!
        #  In a diagonal RK3 method, only 3 gridfunctions need be defined. Below implements this approach.
        y_n_gfs = sp.Symbol("y_n_gfsL", real=True)
        k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs = sp.Symbol("k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfsL", real=True)
        k2_or_y_nplus_a32_k2_gfs = sp.Symbol("k2_or_y_nplus_a32_k2_gfsL", real=True)
        # k_1
        body += """
    // In a diagonal RK3 method like this one, only 3 gridfunctions need be defined. Below implements this approach.
    // Using y_n_gfs as input, k1 and apply boundary conditions\n"""
        body += single_RK_substep_input_symbolic(
            comment_block="""// -={ START k1 substep }=-
    // RHS evaluation:
    //  1. We will store k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs now as
    //     ...  the update for the next rhs evaluation y_n + a21*k1*dt
    // Post-RHS evaluation:
    //  1. Apply post-RHS to y_n + a21*k1*dt""",
            substep_time_offset_dt=Butcher[0][0],
            RHS_str=RHS_string,
            RHS_input_str=y_n_gfs,
            RHS_output_str=k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs,
            RK_lhs_list=[k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs],
            RK_rhs_list=[Butcher[1][1]*k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs*dt + y_n_gfs],
            post_RHS_list=[post_RHS_string], post_RHS_output_list=[k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs],
            enable_SIMD=enable_SIMD, gf_aliases=gf_aliases,
            post_post_RHS_string=post_post_RHS_string) + "// -={ END k1 substep }=-\n\n"

        # k_2
        body += single_RK_substep_input_symbolic(
            comment_block="""// -={ START k2 substep }=-
    // RHS evaluation:
    //    1. Reassign k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs to be the running total y_{n+1}; a32*k2*dt to the running total
    //    2. Store k2_or_y_nplus_a32_k2_gfs now as y_n + a32*k2*dt
    // Post-RHS evaluation:
    //    1. Apply post-RHS to both y_n + a32*k2 (stored in k2_or_y_nplus_a32_k2_gfs)
    //       ... and the y_{n+1} running total, as they have not been applied yet to k2-related gridfunctions""",
            RHS_str=RHS_string,
            substep_time_offset_dt=Butcher[1][0],
            RHS_input_str=k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs,
            RHS_output_str=k2_or_y_nplus_a32_k2_gfs,
            RK_lhs_list=[k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs, k2_or_y_nplus_a32_k2_gfs],
            RK_rhs_list=[Butcher[3][1]*(k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs - y_n_gfs)/Butcher[1][1] + y_n_gfs +␣
↪Butcher[3][2]*k2_or_y_nplus_a32_k2_gfs*dt,
                         Butcher[2][2]*k2_or_y_nplus_a32_k2_gfs*dt + y_n_gfs],
            post_RHS_list=[post_RHS_string, post_RHS_string],
            post_RHS_output_list=[k2_or_y_nplus_a32_k2_gfs,
                                  k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs],
            enable_SIMD=enable_SIMD, gf_aliases=gf_aliases,
            post_post_RHS_string=post_post_RHS_string) + "// -={ END k2 substep }=-\n\n"

        # k_3
        body += single_RK_substep_input_symbolic(
            comment_block="""// -={ START k3 substep }=-
        // RHS evaluation:
        //    1. Add k3 to the running total and save to y_n
        // Post-RHS evaluation:
        //    1. Apply post-RHS to y_n""",
            substep_time_offset_dt=Butcher[2][0],
            RHS_str=RHS_string,
            RHS_input_str=k2_or_y_nplus_a32_k2_gfs, RHS_output_str=y_n_gfs,
            RK_lhs_list=[y_n_gfs],
            RK_rhs_list=[k1_or_y_nplus_a21_k1_or_y_nplus1_running_total_gfs + Butcher[3][3]*y_n_gfs*dt],
            post_RHS_list=[post_RHS_string],
            post_RHS_output_list=[y_n_gfs],
            enable_SIMD=enable_SIMD, gf_aliases=gf_aliases,
            post_post_RHS_string=post_post_RHS_string) + "// -={ END k3 substep }=-\n\n"
    else:
        y_n = sp.Symbol("y_n_gfsL", real=True)
        if not diagonal(MoL_method):
            for s in range(num_steps):
                next_y_input = sp.Symbol("next_y_input_gfsL", real=True)

                # If we're on the first step (s=0), we use y_n gridfunction as input.
                #      Otherwise next_y_input is input. Output is just the reverse.
                if s == 0:  # If on first step:
                    RHS_input = y_n
                else:  # If on second step or later:
                    RHS_input = next_y_input
                RHS_output = sp.Symbol("k" + str(s + 1) + "_gfs", real=True)
                if s == num_steps - 1:  # If on final step:
                    RK_lhs = y_n
                else:  # If on anything but the final step:
                    RK_lhs = next_y_input
                RK_rhs = y_n
                for m in range(s + 1):
                    k_mp1_gfs = sp.Symbol("k" + str(m + 1) + "_gfsL")
                    if Butcher[s + 1][m + 1] != 0:
                        if Butcher[s + 1][m + 1] != 1:
                            RK_rhs += dt * k_mp1_gfs*Butcher[s + 1][m + 1]
                        else:
                            RK_rhs += dt * k_mp1_gfs

                post_RHS = post_RHS_string
                if s == num_steps - 1:  # If on final step:
                    post_RHS_output = y_n
                else:  # If on anything but the final step:
                    post_RHS_output = next_y_input

                body += single_RK_substep_input_symbolic(
                    comment_block="// -={ START k" + str(s + 1) + " substep }=-",
                    substep_time_offset_dt=Butcher[s][0],
                    RHS_str=RHS_string,
                    RHS_input_str=RHS_input, RHS_output_str=RHS_output,
                    RK_lhs_list=[RK_lhs], RK_rhs_list=[RK_rhs],
                    post_RHS_list=[post_RHS],
                    post_RHS_output_list=[post_RHS_output],
                    enable_SIMD=enable_SIMD, gf_aliases=gf_aliases,
                    post_post_RHS_string=post_post_RHS_string) + "// -={ END k" + str(s + 1) + " substep }=-\n\n"
        else:
            y_n = sp.Symbol("y_n_gfsL", real=True)
            y_nplus1_running_total = sp.Symbol("y_nplus1_running_total_gfsL", real=True)
            if MoL_method == 'Euler':  # Euler's method doesn't require any k_i, and gets its own unique algorithm
                body += single_RK_substep_input_symbolic(
                    comment_block=indent + "// ***Euler timestepping only requires one RHS evaluation***",
                    substep_time_offset_dt=Butcher[0][0],
                    RHS_str=RHS_string,
                    RHS_input_str=y_n, RHS_output_str=y_nplus1_running_total,
                    RK_lhs_list=[y_n], RK_rhs_list=[y_n + y_nplus1_running_total*dt],
                    post_RHS_list=[post_RHS_string],
                    post_RHS_output_list=[y_n],
                    enable_SIMD=enable_SIMD, gf_aliases=gf_aliases,
                    post_post_RHS_string=post_post_RHS_string)
            else:
                for s in range(num_steps):
                    # If we're on the first step (s=0), we use y_n gridfunction as input.
                    # and k_odd as output.
                    if s == 0:
                        RHS_input  = sp.Symbol("y_n_gfsL", real=True)
                        RHS_output = sp.Symbol("k_odd_gfsL", real=True)
                    # For the remaining steps the inputs and ouputs alternate between k_odd and k_even
                    elif s % 2 == 0:
                        RHS_input = sp.Symbol("k_even_gfsL", real=True)
                        RHS_output = sp.Symbol("k_odd_gfsL", real=True)
                    else:
                        RHS_input = sp.Symbol("k_odd_gfsL", real=True)
                        RHS_output = sp.Symbol("k_even_gfsL", real=True)
                    RK_lhs_list = []
                    RK_rhs_list = []
                    if s != num_steps-1:  # For anything besides the final step
                        if s == 0:  # The first RK step
                            RK_lhs_list.append(y_nplus1_running_total)
                            RK_rhs_list.append(RHS_output*dt*Butcher[num_steps][s+1])

                            RK_lhs_list.append(RHS_output)
                            RK_rhs_list.append(y_n + RHS_output*dt*Butcher[s+1][s+1])
                        else:
                            if Butcher[num_steps][s+1] != 0:
                                RK_lhs_list.append(y_nplus1_running_total)
                                if Butcher[num_steps][s+1] != 1:
                                    RK_rhs_list.append(y_nplus1_running_total + RHS_output*dt*Butcher[num_steps][s+1])
                                else:
                                    RK_rhs_list.append(y_nplus1_running_total + RHS_output*dt)
                            if Butcher[s+1][s+1] != 0:
                                RK_lhs_list.append(RHS_output)
                                if Butcher[s+1][s+1] != 1:
                                    RK_rhs_list.append(y_n + RHS_output*dt*Butcher[s+1][s+1])
                                else:
                                    RK_rhs_list.append(y_n + RHS_output*dt)
                        post_RHS_output = RHS_output
                    if s == num_steps-1:  # If on the final step
                        if Butcher[num_steps][s+1] != 0:
                            RK_lhs_list.append(y_n)
                            if Butcher[num_steps][s+1] != 1:
                                RK_rhs_list.append(y_n + y_nplus1_running_total + RHS_output*dt*Butcher[num_steps][s+1])
                            else:
                                RK_rhs_list.append(y_n + y_nplus1_running_total + RHS_output*dt)
                        post_RHS_output = y_n
                    body += single_RK_substep_input_symbolic(
                        comment_block=indent + "// -={ START k" + str(s + 1) + " substep }=-",
                        substep_time_offset_dt=Butcher[s][0],
                        RHS_str=RHS_string,
                        RHS_input_str=RHS_input, RHS_output_str=RHS_output,
                        RK_lhs_list=RK_lhs_list, RK_rhs_list=RK_rhs_list,
                        post_RHS_list=[post_RHS_string],
                        post_RHS_output_list=[post_RHS_output],
                        enable_SIMD=enable_SIMD, gf_aliases=gf_aliases,
                        post_post_RHS_string=post_post_RHS_string) + "// -={ END k" + str(s + 1) + " substep }=-\n\n"
```

```
    body += """
// To minimize roundoff error (from adding dt to params.time lots of times),
//   here we set time based on the iteration number.
griddata->params.time = (REAL)(griddata->params.n + 1) * griddata->params.dt;

// Finally, increment the timestep n:
griddata->params.n++;
"""
    enableCparameters=False
    add_to_Cfunction_dict(
        includes=includes,
        desc=desc,
        c_type=c_type, name=name, params=params,
        body=indent_Ccode(body, "  "),
        enableCparameters=enableCparameters, rel_path_to_Cparams=os.path.join("."))
```

3.4 Step 3.d: Memory deallocation: `MoL_free_memory()` [Back to top]

We define the function `MoL_free_memory()` which generates the C code for freeing the memory that was being occupied by the grid functions lists that had been allocated.

[7]:
```
# add_to_Cfunction_dict_MoL_free_memory() registers
#           MoL_free_memory_y_n_gfs() and
#           MoL_free_memory_non_y_n_gfs(), which free memory for
#           the indicated sets of gridfunctions
def add_to_Cfunction_dict_MoL_free_memory(MoL_method, which_gfs):
    includes = ["NRPy_basic_defines.h", "NRPy_function_prototypes.h"]
    desc = "Method of Lines (MoL) for \"" + MoL_method + "\" method: Free memory for \"" + which_gfs + "\" gridfunctions\n"
    desc += "   - y_n_gfs are used to store data for the vector of gridfunctions y_i at t_n, at the start of each MoL timestep\n"
    desc += "   - non_y_n_gfs are needed for intermediate (e.g., k_i) storage in chosen MoL method\n"
    c_type = "void"

    y_n_gridfunctions, non_y_n_gridfunctions_list, _diagnostic_gridfunctions_point_to, \
        _diagnostic_gridfunctions2_point_to = generate_gridfunction_names(MoL_method=MoL_method)

    if which_gfs == "y_n_gfs":
        gridfunctions_list = [y_n_gridfunctions]
    elif which_gfs == "non_y_n_gfs":
        gridfunctions_list = non_y_n_gridfunctions_list
    else:
        print("ERROR: which_gfs = \"" + which_gfs + "\" unrecognized.")
        sys.exit(1)
    name = "MoL_free_memory_" + which_gfs
    params = "const paramstruct *restrict params, MoL_gridfunctions_struct *restrict gridfuncs"
    body = ""
    for gridfunctions in gridfunctions_list:
        body += "    free(gridfuncs->" + gridfunctions + ");\n"
    add_to_Cfunction_dict(
        includes=includes,
        desc=desc,
        c_type=c_type, name=name, params=params,
        body=indent_Ccode(body, "  "),
        rel_path_to_Cparams=os.path.join("."))
```

3.5 Step 3.e: Define & register `MoL_gridfunctions_struct` in `NRPy_basic_defines.h`: `NRPy_basic_defines_MoL_timestepping_struct()` [Back to top]

`MoL_gridfunctions_struct` stores pointers to all the gridfunctions needed by MoL, and we define this struct within `NRPy_basic_defines.h`.

[8]:
```
# Register MoL_gridfunctions_struct in NRPy_basic_defines
def NRPy_basic_defines_MoL_timestepping_struct(MoL_method="RK4"):
    y_n_gridfunctions, non_y_n_gridfunctions_list, _diagnostic_gridfunctions_point_to, \
        _diagnostic_gridfunctions2_point_to = generate_gridfunction_names(MoL_method=MoL_method)
    # Step 3.b: Create MoL_timestepping struct:
    indent = "  "
    Nbd = "typedef struct __MoL_gridfunctions_struct__ {\n"
    Nbd += indent + "REAL *restrict " + y_n_gridfunctions + ";\n"
    for gfs in non_y_n_gridfunctions_list:
        Nbd += indent + "REAL *restrict " + gfs + ";\n"
    Nbd += indent + "REAL *restrict diagnostic_output_gfs;\n"
    Nbd += indent + "REAL *restrict diagnostic_output_gfs2;\n"
    Nbd += "} MoL_gridfunctions_struct;\n"
    Nbd += """#define LOOP_ALL_GFS_GPS(ii) _Pragma("omp parallel for") \\
  for(int (ii)=0;(ii)<Nxx_plus_2NGHOSTS0*Nxx_plus_2NGHOSTS1*Nxx_plus_2NGHOSTS2*NUM_EVOL_GFS;(ii)++)\n"""

    outC_NRPy_basic_defines_h_dict["MoL"] = Nbd
```

3.6 Step 3.f: Add all MoL C codes to C function dictionary, and add MoL definitions to `NRPy_basic_defines.h`: `register_C_functions_and_NRPy_basic_defines()` [Back to top]

[9]:
```
# Finally declare the master registration function
def register_C_functions_and_NRPy_basic_defines(MoL_method = "RK4",
            RHS_string =  "rhs_eval(Nxx,Nxx_plus_2NGHOSTS,dxx, RK_INPUT_GFS, RK_OUTPUT_GFS);",
            post_RHS_string = "apply_bcs(Nxx,Nxx_plus_2NGHOSTS, RK_OUTPUT_GFS);", post_post_RHS_string = "",
            enable_rfm=False, enable_curviBCs=False, enable_SIMD=False):
    for which_gfs in ["y_n_gfs", "non_y_n_gfs"]:
        add_to_Cfunction_dict_MoL_malloc(MoL_method, which_gfs)
        add_to_Cfunction_dict_MoL_free_memory(MoL_method, which_gfs)
    add_to_Cfunction_dict_MoL_step_forward_in_time(MoL_method, RHS_string, post_RHS_string, post_post_RHS_string,
                                                   enable_rfm=enable_rfm, enable_curviBCs=enable_curviBCs,
                                                   enable_SIMD=enable_SIMD)
    NRPy_basic_defines_MoL_timestepping_struct(MoL_method=MoL_method)
```

4 Step 4: Code Validation against `MoLtimestepping.MoL` NRPy+ module [Back to top]

As a code validation check, we verify agreement in the dictionary of Butcher tables between

1. this tutorial and
2. the NRPy+ MoLtimestepping.MoL module.

We generate the header files for each RK method and check for agreement with the NRPy+ module.

[11]:
```
import sys
import MoLtimestepping.MoL as MoLC
import difflib
import pprint
# Courtesy https://stackoverflow.com/questions/12956957/print-diff-of-python-dictionaries ,
#   which itself is an adaptation of some Cpython core code
def compare_dicts(d1, d2):
    return ('\n' + '\n'.join(difflib.ndiff(
                   pprint.pformat(d1).splitlines(),
                   pprint.pformat(d2).splitlines())))


print("\n\n ### BEGIN VALIDATION TESTS ###")
for key, value in Butcher_dict.items():
    if key not in {"AHE", "ABS", "ARKF", "ACK", "ADP5", "ADP8", "AB"}:
        # This validation does not work on anything other than standard RK methods, so they are excluded.
        register_C_functions_and_NRPy_basic_defines(key,
                                        "rhs_eval(Nxx,Nxx_plus_2NGHOSTS,dxx, RK_INPUT_GFS, RK_OUTPUT_GFS);",
                                       "apply_bcs(Nxx,Nxx_plus_2NGHOSTS, RK_OUTPUT_GFS);")
        from outputC import outC_function_dict, outC_function_master_list
        notebook_dict = outC_function_dict.copy()
        notebook_master_list = list(outC_function_master_list)
        outC_function_dict.clear()
        del outC_function_master_list[:]
        from outputC import outC_function_dict
        if outC_function_dict != {}:
            print("Error in clearing outC_function_dict.")
            sys.exit(1)
        MoLC.register_C_functions_and_NRPy_basic_defines(key,
                                "rhs_eval(Nxx,Nxx_plus_2NGHOSTS,dxx, RK_INPUT_GFS, RK_OUTPUT_GFS);",
                               "apply_bcs(Nxx,Nxx_plus_2NGHOSTS, RK_OUTPUT_GFS);")
        from outputC import outC_function_dict
        python_module_dict = outC_function_dict

        if notebook_dict != python_module_dict:
            print("VALIDATION TEST FAILED.\n")
            print(compare_dicts(notebook_dict, python_module_dict))
            sys.exit(1)
        print("VALIDATION TEST PASSED on all files from "+str(key)+" method")
print("### END VALIDATION TESTS ###")
```

```
 ### BEGIN VALIDATION TESTS ###
VALIDATION TEST PASSED on all files from Euler method
VALIDATION TEST PASSED on all files from RK2 Heun method
VALIDATION TEST PASSED on all files from RK2 MP method
VALIDATION TEST PASSED on all files from RK2 Ralston method
VALIDATION TEST PASSED on all files from RK3 method
VALIDATION TEST PASSED on all files from RK3 Heun method
VALIDATION TEST PASSED on all files from RK3 Ralston method
VALIDATION TEST PASSED on all files from SSPRK3 method
VALIDATION TEST PASSED on all files from RK4 method
VALIDATION TEST PASSED on all files from DP5 method
VALIDATION TEST PASSED on all files from DP5alt method
VALIDATION TEST PASSED on all files from CK5 method
VALIDATION TEST PASSED on all files from DP6 method
VALIDATION TEST PASSED on all files from L6 method
VALIDATION TEST PASSED on all files from DP8 method
### END VALIDATION TESTS ###
```

5 Step 5: Output this notebook to LaTeX-formatted PDF [Back to top]

The following code cell converts this Jupyter notebook into a proper, clickable LaTeX-formatted PDF file. After the cell is successfully run, the generated PDF may be found in the root NRPy+ tutorial directory, with filename `Tutorial-RK_Butcher_Table_Generating_C_Code.pdf`. (Note that clicking on this link may not work;

you may need to open the PDF file through another means.)

```
[ ]: import cmdline_helper as cmd    # NRPy+: Multi-platform Python command-line interface
     cmd.output_Jupyter_notebook_to_LaTeXed_PDF("Tutorial-Method_of_Lines-C_Code_Generation")
```