JFNK Documentation

Release

Namdi Brandon

Jul 11, 2018

Contents:

This module contains code for functions for solving temporal ordinary differential equations (ODEs)

$$\frac{dy(t)}{dt} = f(t, y(t))$$
$$y(0) = y_0$$

which has the following solution

$$y(t) = y_0 + \int_0^t f(\tau, y(\tau))\,\mathrm{d}\tau$$

The numerical integrators are the following:

1. Backward (implicit) Euler method
2. Jacobian-Free Newton-Krylov (JFNK) method with uniform time stepping
3. Jacobian-Free Newton-Krylov (JFNK) method with adaptive time stepping
4. Spectral Deferred Corrections (SDC) (implicit)
5. Spectral solution (Gauss collocation formulation) solver

The JFNK solver is designed to efficiently solve **stiff systems**. The ideas behind the implementation of the method are mentioned in *[QBC+16]*.

CHAPTER 1

Source Files

This directory contains the files necessary for solving the differential equation system. Namely, the numerical integrators are contained in the `integrator.py` file and the input parameters are in the `params.py` file.

1.1 analysis module

This module contains class `analysis.Result` that help saves the information related to the ODE solution. Also, this module contains functions that aid in analyzing the results of the various solutions.

class analysis.**Result**(*p*, *t0*, *t_final*, *t*, *y*, *is_adaptive*, *Y=None*, *D=None*, *h=None*, *dt_init=None*, *be_tol=None*, *sdc_tol=None*, *tol=None*)

Bases: `object`

This class saves the information related to the ODE solution and the parameters used in order to generate the solution.

Parameters

- **p** (`params.Params`) – the parameters related to the solver
- **t0** (*float*) – the initial time t_{init}
- **t_final** (*float*) – the final time t_{final}
- **t** (*numpy.ndarray*) – the time nodes used in the simulation
- **y** (*numpy.ndarray*) – the approximation at the time nodes
- **is_adaptive** (*bool*) – a flag indicating whether the simulation used adaptive (if True) step sizes or uniform step sizes (if False)
- **Y** – a history of the solution history for each iteration at each time step
- **D** – a history of the deferred correction for each iteration at each time step
- **h** (*float*) – the adaptive time steps used
- **dt_init** (*float*) – the initial step size Δt_{init} used in the adaptive scheme

- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **tol** (*float*) – the approximated absolute error at each step for the adaptive solution

analysis.**analyze_corrections** (*D*, *Y*)

This function calculates the following

1. for each iteration k, the magnitude of the correction for each component i: $||\delta_i^{[k]}||$
2. for each iteration k, the magnitude of the correction for each component i on a log_{10} scale : $log_{10}\left(||\delta_i^{[k]}||\right)$
3. for each iteration k, the magnitude of the relative correction for each component i: $\frac{||\delta_i^{[k]}||}{||y_i^{[k]}||}$
4. for each iteration k, the magnitude of the relative correction for each component i on a log_{10} scale: $log_{10}\left(\frac{||\delta_i^{[k]}||}{||y_i^{[k]}||}\right)$

Parameters

- **D** (*list*) – corrections, dimensions (n iterations, p nodes, m problem size)
- **Y** (*list*) – approximations, dimensions (n iterations, p nodes, m problem size)

Returns for each iteration the magnitude of the correction for each component, for each iteration the magnitude of the correction for each component in log base 10, for each iteration the magnitude of the relative correction for each component, for each iteration the magnitude of the relative correction for each component in log base 10

Return type numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray

analysis.**error_analysis** (*y_approx*, *y_solution*, *threshold=1e-20*)

This function calculates the absolute error or the relative error.

Parameters

- **y_approx** (*numpy.ndarray*) – the approximate solution
- **y_solution** (*numpy.ndarray*) – the more accurate solution
- **threshold** (*float*) – the threshold to set the components to 0

Returns the absolute error, the relative error

Return type numpy.ndarray, numpy.ndarray

analysis.**get_correction_norms** (*Y*, *D*)

This function calculates various measures of the corrections from SDC sweep

1. for each iteration k, the maximum magnitude of the relative correction for each component i: $\max \frac{||\delta_i^{[k]}||}{||y_i^{[k]}||}$
2. for each iteration k, the maximum magnitude of the correction for each component i: $\max ||\delta_i^{[k]}||$
3. for each iteration k, the mean magnitude of the relative correction for each component i: $E\left[\frac{||\delta_i^{[k]}||}{||y_i^{[k]}||}\right]$
4. for each iteration k, the mean magnitude of the correction for each component i: $E[||\delta_i^{[k]}||]$

Parameters

- **Y** (*list*) – the approximations for the solution at each iteration arrays of dimensions (n nodes, size of problem)

- **D** (*list*) – the corrections for each iteration arrays of dimensions (n nodes, size of problem)

Returns the maximum relative norm, the maximum absolute norm, the mean relative norm, the mean absolute norm

Return type numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray

`analysis.`**`get_error_time_nodes`**(*t*, *y*, *y_spline*, *do_relative=True*)

This function calculates the absolute or relative error comparing the approximate solution and the "exact" solution at all of the temporal nodes.

Parameters

- **t** (*numpy.ndarray*) – the temporal nodes on the approximate solution
- **y** (*numpy.ndarray*) – the approximate solution
- **y_spline** (*function*) – a function that may interpolate the "exact" (more accurate) solution
- **do_relative** (*bool*) – a flag indicating whether (if True) to calculate the relative error or not to (if False) to calculate the absolute error

Returns the absolute or relative errors

Return type numpy.ndarray

`analysis.`**`get_error_time_steps`**(*t*, *y*, *y_spline*, *n_nodes*, *do_relative=True*)

This function calculates the absolute or relative error comparing the approximate solution and the "exact" solution at the end of each time step.

Parameters

- **t** (*numpy.ndarray*) – the temporal nodes
- **y** (*numpy.ndarray*) – the approximate solution
- **y_spline** (*function*) – a function that may interpolate the "exact" (more accurate) solution
- **n_nodes** (*int*) – the number of nodes per time step
- **do_relative** (*bool*) – a flag indicating whether (if True) to calculate the relative error or not to (if False) to calculate the absolute error

Returns the absolute or relative error

Return type numpy.ndarray

`analysis.`**`relative_norm`**(*top*, *bot*)

This function calculates the relative ratios between norms of vectors. Given two sets of vectors

$$top_{n\times m}, bot_{n\times m}$$

where n is the number of temporal nodes and m is the size of the system.

For each component j, calculate the relative norms over the time nodes

$$ratio_j = \frac{||top_j||_2}{||bot_j||_2}$$

Make sure, we avoid division by zero

$$x_j = \begin{cases} ratio_j & \text{if } ||bot_j||_2 \neq 0 \\ ||top_j||_2 & \text{if } ||bot_j||_2 = 0 \end{cases}$$

Return the maximum value

$$\|x\|_\infty$$

Parameters

- **`top`** – the top vector for a given iteration dimensions (n nodes, m size of problem)
- **`bot`** – the bottom vector for a given iteration dimensions (n nodes, m size of problem)

Returns the maximum value of the relative norm between two vectors

Return type float

`analysis.`**`run_threshold`**`(`*y*, *threshold*`)`

If the value of the solution is below the threshold, set it to zero.

Parameters

- **`y`** (*`numpy.ndarray`*) – the approximate solution
- **`threshold`** (*`float`*) – the threshold

Returns None

1.2 integrator module

This module contains code for functions for solving temporal ordinary differential equations (ODEs)

$$\begin{aligned}\frac{dy(t)}{dt} &= f(t, y(t)) \\ y(0) &= y_0\end{aligned}$$

which has the following solution

$$y(t) = y_0 + \int_0^t f(\tau, y(\tau))\,\mathrm{d}\tau$$

The numerical integrators are the following:

- Backward (implicit) Euler method
- Jacobian-Free Newton-Krylov (JFNK) method with uniform time stepping
- Jacobian-Free Newton-Krylov (JFNK) method with adaptive time stepping
- Spectral Deferred Corrections (SDC) (implicit)
- Spectral solution (Gauss collocation formulation) solver

Abbreviations	Meaning
JFNK	Jacobian-Free Newton Krylov
ODE	Ordinary differential equation
SDC	Spectral deferred corrections

`integrator.`**`adjust_scaler`**`(`*x*, *x_min=1.5*, *x_max=4*`)`

For the adaptive time stepping algorithm, this function adjusts the adaptive step size

$$x \leftarrow \begin{cases} x & \text{if } x < 1 \\ 1 & \text{if } 1 \le x \le x_{min} \\ \min(x, x_{max}) & \text{if } x > x_{min} \end{cases}$$

Parameters

- **x** (*float*) – the ratio in which to grow or shrink the time step
- **x_min** (*float*) – the minimum ratio in which to grow a time step
- **x_max** (*float*) – the maximum ratio in which to grow a time step

Returns the ratio in which to grow or shrink the time step

Return type float

`integrator.`**`backward_euler`**(*f_eval*, *t*, *y0*, *S*, *be_tol=1e-12*, *do_print=False*)

This function runs the backward Euler method over all of the nodes t_i in an entire time step of size Δt.

$$y - \Delta t\tilde{S}F(y) = rhs$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*numpy.ndarray*) – the time nodes over a time step Δt
- **y0** (*numpy.ndarray*) – the initial condition (length = m)
- **S** (*numpy.ndarray*) – the backward Euler integration matrix $\tilde{S}$

Returns

`integrator.`**`backward_euler_node`**(*f_eval*, *t*, *h*, *rhs*, *x0*, *be_tol=1e-12*)

This function solves an ODE system using backward Euler method at a specific time t. This code uses a general numerical solver in order to do the inversion in the backward Euler method in order to find y_i.

$$y_i - h_i f(t_i, y_{i-1}) = rhs_i$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*float*) – the time at a node
- **h** (*float*) – the step size
- **rhs** (*numpy.ndarray*) – the right hand side of the backward euler system [m x 1]
- **x0** (*numpy.ndarray*) – the initial guess for the solution [m x 1]
- **be_tol** (*float*) – the tolerance for the backward Euler solver

Returns the approximate solution [m x 1]

Return type numpy.ndarray

`integrator.`**`convergence_criteria`**(*d*, *y*, *tol*)

This function calculates whether or not the correction vector δ is small enough to satisfy the convergence criteria.

$$x = \frac{||\delta||}{||y||}$$

$$\begin{cases} x \le tol & \text{converged} \\ x > tol & \text{not converged} \end{cases}$$

Parameters

- **d** (*numpy.ndarray*) – the correction vector δ for a given iteration dimensions (n nodes, size of problem)
- **y** (*numpy.ndarray*) – the approximate solution y for a given iteration dimensions (n nodes, size of problem)
- **tol** (*float*) – the correction tolerance for the convergence criteria

Returns a flag indicating whether or not the corrections are small enough to qualify for convegence

Return type bool

`integrator.`**`jfnk`**(*f_eval*, *t*, *y0*, *y_approx*, *S*, *S_p*, *spectral_radius*, *n_iter_max_newton=50*, *be_tol=1e-12*, *sdc_tol=1e-14*, *do_print=False*)

The Jacobian-Free Newton-Krylov (JFNK) method to approximate a solution to the spectral solution

$$y - \Delta t S F(y) = y_0$$

over one time step of size Δt. This is done by use a modified version of Newton's method to find a calculate a solution

$$H(y) = 0$$

where $H(y^{[k]}) = \delta^{[k]}$ corresponds to one iteration of the SDC method.

Given $y^{[0]}$, this method does the following

1. calculate the initial SDC iterations

$$\begin{cases} \delta^{[k]} & = H(y^{[k]}) \text{calculate an SDC correction} \\ y^{[k+1]} & = y^{[k]} + \delta^{[k]} \text{update the SDC solution} \end{cases}$$

   until the the solution converges or order convergence has been observed

2. do the Newton (Jacobian-Free) iterations

$$\begin{aligned} & J_H(y^{[p]})\Delta x = -H(y^{[p]}) \\ \implies & J_H(y^{[p]})\Delta x = -\delta^{[p]} \end{aligned}$$

   Set $\Delta x = \sum_{j=0}^{p-1} c_j \delta^{[j]}$ and solve

$$\begin{aligned} & J_H(y^{[p]}) \sum_{j=0}^{p-1} c_j \delta^{[j]} = -\delta^{[p]} \\ \implies & \sum_{j=0}^{p-1} c_j (\delta^{[k+1]} - \delta^{[k]}) = -\delta^{[p]} \end{aligned}$$

   Solve the system for the Jacobian-Free system

$$\begin{cases} Ac & = -\delta^{[p]} \\ y & \leftarrow y^{[p]} + \sum_{j=0}^{p-1} c_j \delta^{[j]} \end{cases}$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*numpy.ndarray*) – the time nodes over a time step Δt
- **y0** (*numpy.ndarray*) – the initial condition (length = m)
- **y_approx** (*numpy.ndarray*) – the provisional solution (dimensions n time nodes, size of the problem)
- **S** (*numpy.ndarray*) – the spectral integration (Gaussian quadrature) matrix, S
- **S_p** (*numpy.ndarray*) – the backward Euler integration matrix, $\tilde{S}$
- **spectral_radius** (*float*) – the spectral radius of the correction matrix for the extremely stiff case
- **n_iter_max_newton** (*int*) – the maximum number of Newton iterations
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **do_print** (*bool*) – a flag indicating whether or not to print the elapsed time

Returns the solution, the history of approximations for each iteration, the history of corrections for each iteration

Return type numpy.ndarray (dimensions n time nodes, size of the problem) , list (length number of iterations), list (length of iterations), bool, bool, numpy.ndarray (length of iterations)

integrator.**jfnk_adaptive**(*f_eval, t_init, t_final, dt_init, p, y0, tol, n_iter_max_newton=50, be_tol=1e-12, sdc_tol=1e-14, n_steps_max=1000000000, do_print=False*)

Run the JFNK with adaptive step sizes from $t \in [t_{init}, t_{final}]$ to calculate an approximation to the solution

$$y(t_{final}) = y(t_{init}) + \int_{t_{init}}^{t_{final}} f(\tau, y(\tau))\, \mathrm{d}\tau$$

Such that for each time step the step size Δt is chosen so that the difference between the exact solution y and the approximate solution $\tilde{y}$

$$|||y - \tilde{y}||_\infty \leq tol$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t_init** (*float*) – the initial time t_{init}
- **t_final** (*float*) – the final time t_{final}
- **dt_init** (*float*) – the initial step size Δt_{init}
- **p** (params.Params) – the parameters related to the solver
- **y0** (*numpy.ndarray*) – the initial condition $y(t_{init})$ (length = m)
- **tol** (*float*) – the approximated absolute error at each step for the adaptive solution
- **n_iter_max_newton** (*int*) – the maximum number of Newton iterations
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **n_steps_max** (*int*) – the maximum number of steps in the solver

- **do_print** (*bool*) – a flag indicating whether or not to print the elapsed time

Returns all of the time nodes, the value of the solution at each node, a history of the solution history for each iteration at each time step, history of the deferred correction for each iteration at each time step, the step size for each time step

Return type numpy.ndarray, numpy.ndarray, list, list, numpy.ndarray

integrator.**jfnk_initial** (*f_eval*, *t*, *y0*, *y_approx*, *S*, *S_p*, *spectral_radius*, *be_tol=1e-12*, *sdc_tol=1e-14*, *n_iter_max=500*, *do_print=False*)

This function runs initial SDC iterations until order convergence is observed. Given $H(y^{[k]}) = \delta^{[k]}$ corresponds to one iteration of the SDC method.

1. Run 2 initial SDC iterations

$$\begin{cases} \delta^{[k]} & \leftarrow H(y^{[k]}) \\ y^{[k+1]} & \leftarrow y^{[k]} + \delta^{[k]}, k = 0, 1 \end{cases}$$

2. Caculate the ratio of the corrections

$$r = \frac{\|\delta^{[k-1]}\|_F}{\|\delta^{[k-2]}\|_F}$$

where $\|\cdot\|_F$ is the Frobenius norm.

If $\frac{r}{\rho(C_s)} > 0.1$, order convergence is observed. Stop the function.

If $\frac{r}{\rho(C_s)} \leq 0.1$, if not converged, do another SDC iteration and go to step 2.

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*numpy.ndarray*) – the time nodes over a time step Δt
- **y0** (*numpy.ndarray*) – the initial condition
- **y_approx** (*numpy.ndarray*) – the provisional solution (dimensions n time nodes, size of the problem)
- **S** – the spectral (Gaussian quadrature) integration matrix
- **S_p** – the backward Euler integration matrix
- **spectral_radius** (*float*) – the spectral radius of the correction matrix for the extremely stiff case
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **n_iter_max** – the maximum number of SDC iterations
- **do_print** (*bool*) – a flag indicating whether or not to print the elapsed time

Returns the approximation, the history of the approximations for each iteration, the history of the deferred corrections for each iteration, a flag indicating whether or not the solution has converged, a flag indicating whether or not the problem is stiff, relative magnitude of consecutive iterations

Return type numpy.ndarray, list, list, bool, bool, numpy.ndarray

integrator.**jfnk_iterations** (*f_eval*, *t*, *y0*, *y_init*, *S*, *S_p*, *n_iter_max_newton=50*, *be_tol=1e-12*, *sdc_tol=1e-14*)

This function solves the Newton's method iterations for solving

$$H(y) = 0$$

where $H(y^{[k]}) = \delta^{[k]}$ corresponds to one iteration of the SDC method.

1. Run $n+1$ SDC iterations.

$$\begin{cases} \delta^{[k]} & \leftarrow H(y^{[k]}) \\ y^{[k+1]} & \leftarrow y^{[k]} + \delta^{[k]}, k = 0, \ldots, n \end{cases}$$

2. Solve the Newton iteration system without using the Jacobian explicitly

$$\begin{aligned} & J_H(y^{[n]})\Delta x = -H(y^{[n]}) \\ \implies & J_H(y^{[n]})\Delta x = -\delta^{[n]} \end{aligned}$$

Set $\Delta x = \sum_{j=0}^{n-1} c_j \delta^{[j]}$ and solve

$$\begin{aligned} & J_H(y^{[n]}) \sum_{j=0}^{n-1} c_j \delta^{[j]} = -\delta^{[n]} \\ \implies & \sum_{j=0}^{n-1} c_j (\delta^{[k+1]} - \delta^{[k]}) = -\delta^{[n]} \end{aligned}$$

Solve the system for the Jacobian-Free system

$$\begin{cases} Ac & = -\delta^{[n]} \\ y & \leftarrow y^{[n]} + \sum_{j=0}^{n-1} c_j \delta^{[j]} \end{cases}$$

3. If not converged, repeat by going to step 1.

Parameters

- **`f_eval`** (*`function`*) – the derivative function $y' = f(t, y)$.
- **`t`** (*`numpy.ndarray`*) – the time nodes in the time step (length n_nodes)
- **`y0`** (*`numpy.ndarray`*) – the initial solution [size of the problem x 1]
- **`y_init`** (*`numpy.ndarray`*) – the approximate solution [n_nodes x size of the problem]
- **`S`** (*`numpy.ndarray`*) – the spectral integration matrix
- **`S_p`** (*`numpy.ndarray`*) – the preconditioner integration matrix
- **`n_iter_max_newton`** (*`int`*) – the maximum number of Newton iterations

Returns the solution, the history of the solutions for each iteration, the deferred correction for each iteration, a flag indicating whether the procedure converged

Return type numpy.ndarray, list, list, bool

`integrator.`**`jfnk_step`**(*f_eval*, *p*, *dt*, *t*, *y0*, *n_iter_max_newton=50*, *be_tol=1e-12*, *sdc_tol=1e-14*, *do_print=False*)

This function runs everything needed for the JFNK to run for 1 time step (i.e., the backward euler precondition and the JFNK iterations).

Parameters

- **`f_eval`** (*`function`*) – the derivative function $y' = f(t, y)$.
- **`p`** (`params.Params`) – the parameters related to the solver
- **`dt`** (*`float`*) – the time step Δt
- **`t`** (*`numpy.ndarray`*) – the number of temporal nodes (length number of time nodes)

- **y0** (*numpy.ndarray*) – the initial condition (length, size of the problem)
- **n_iter_max_newton** – the maximum number of Newton iteration
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **do_print** (*bool*) – a flag indicating whether or not to print the elapsed time

Returns the approximation, the history of the approximations for each iteration, the history of the deferred corrections for each iteration, a flag indicating whether or not the solution has converged, a flag indicating whether or not the problem is stiff, relative magnitude of consecutive iterations

Return type numpy.ndarray, list, list, bool, bool, numpy.ndarray

`integrator.`**`jfnk_uniform`**(*f_eval*, *t_init*, *t_final*, *n_steps*, *p*, *y0*, *n_iter_max_newton=50*, *be_tol=1e-12*, *sdc_tol=1e-14*, *do_print=False*)

Run the JFNK with uniform step sizes from $t \in [t_{init}, t_{final}]$ to calculate an approximation to the solution

$$y(t_{final}) = y(t_{init}) + \int_{t_{init}}^{t_{final}} f(\tau, y(\tau))\,\mathrm{d}\tau$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t_init** (*float*) – the initial time t_{init}
- **t_final** (*float*) – the final time t_{final}
- **nsteps** (*int*) – the number of steps
- **p** (params.Params) – the parameters related to the solver
- **y0** (*numpy.ndarray*) – the initial condition $y(t_{init})$ (length = m)
- **n_iter_max_newton** (*int*) – the maximum number of Newton iterations
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **do_print** (*bool*) – a flag indicating whether or not to print the elapsed time

Returns all of the time nodes, the value of the solution at each node, a history of the solution history for each iteration at each time step, history of the deferred correction for each iteration at each time step

Return type numpy.ndarray, numpy.ndarray, list, list

`integrator.`**`print_elapsed_time`**(*start*, *end*)

This function prints the elapsed time [s] between to time points.

Parameters

- **start** (*float*) – the start time [s]
- **end** (*float*) – the end time [s]

Returns

`integrator.`**`relative_norm`**(*top*, *bot*)

This function calculates the relative ratios between norms of vectors. Given two sets of vectors

$$top_{n \times m}, bot_{n \times m}$$

where n is the number of temporal nodes and m is the size of the system.

For each component j, calculate the relative norms over the time nodes

$$ratio_j = \frac{\|top_j\|_2}{\|bot_j\|_2}$$

Make sure, we avoid division by zero

$$x_j = \begin{cases} ratio_j & \text{if } \|bot_j\|_2 \neq 0 \\ \|top_j\|_2 & \text{if } \|bot_j\|_2 = 0 \end{cases}$$

Return the maximum value

$$\|x\|_\infty$$

Parameters

- **top** – the top vector for a given iteration dimensions (n nodes, m size of problem)
- **bot** – the bottom vector for a given iteration dimensions (n nodes, m size of problem)

Returns the maximum value of the relative norm between two vectors

Return type float

`integrator.`**`run_spectral`**(*f_eval*, *t*, *y0*, *S*, *tol*, *y_approx*, *do_print=False*, *verbose=False*)

`integrator.`**`scaler_lobatto`**(*aerr*, *n_nodes*, *k*, *tol*)

This function calculates x the amount the step size should increase or decrease for the proper adaptive time step size for Gauss-Lobatto nodes.

Parameters

- **aerr** (*numpy.ndarray*) – the maximum absolute error between the approximate solution y_h and the higher accuracy solution $y_{h/k}$
- **n_nodes** (*int*) – the number of temporal nodes within the time step
- **k** (*int*) – the amount of mini time steps leading up to $y(\Delta t)$
- **tol** (*float*) – the approximated absolute error at each step for the adaptive solution

$$x = \left(tol \frac{1 - (\frac{1}{k})^p}{\|y_h - y_{h/k}\|_\infty} \right)^{1/p}$$

where $p = 2 * n - 1$ and n is the number of Gauss-Lobatto nodes

Returns the amount the current step size should increase or decrease for the proper adaptive time step

Return type float

`integrator.`**`sdc`**(*f_eval*, *t*, *y0*, *y_old*, *S*, *S_p*, *n_iter_max_sdc=500*, *be_tol=1e-12*, *sdc_tol=1e-14*, *do_print=False*)

This function runs the SDC method until convergence.

$$\begin{cases} \delta^{[k]} & \leftarrow H(y^{[k]}) \\ y^{[k+1]} & \leftarrow y^{[k]} + \delta^{[k]} \end{cases}$$

where $H(y^{[k]}) = \delta^{[k]}$ corresponds to one iteration of the SDC method.

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*numpy.ndarray*) – the temporal nodes (length number of nodes) over the time step of size Δt
- **y0** (*numpy.ndarray*) – the initial condition $y(t_{init})$ (length = size of the problem)
- **y_old** (*numpy.ndarray*) – the provisional solution (dimensions, size of the problem)
- **S** (*numpy.ndarray*) – the spectral integration (Gaussian quadrature) matrix, S
- **S_p** (*numpy.ndarray*) – the backward Euler integration matrix, $\tilde{S}$
- **n_iter_max_sdc** (*int*) – the maximum number of SDC iterations
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver
- **sdc_tol** (*float*) – the convergence criteria for the SDC solver
- **do_print** (*bool*) – a flag indicating whether or not to print the elapsed time

Returns the SDC solution, a history of the approximate solution for each iteration, the history of the deferred correction for each itearation, a flag indicating whether or not the solution has met the convergence criteria

Return type numpy.ndarray, list, list, bool

`integrator.`**`sdc_node`**(*f_eval*, *t*, *h*, *rhs*, *y_old*, *be_tol=1e-12*)

This function runs SDC on the i^{th} time node within the time step.

$$y_i^{[k+1]} - hF(y_i^{[k+1]}) = rhs_i$$

where

$$rhs_i = y_{i-1}^{[k]} + (\Delta t S_i - he_i) \cdot F(y^{[k]})$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*float*) – the time
- **h** (*float*) – the time step size
- **rhs** (*numpy.ndarray*) – the right hand side of the backward euler system [m x 1]
- **y_old** (*numpy.ndarray*) – the approximate solution before the update [number of nodes, size of the problem)
- **be_tol** (*float*) – the convergence criteria for the backward Euler solver

Returns both the improved solutionand the correction at the time t

Return type numpy.ndarray, numpy.ndarray

`integrator.`**`sdc_sweep`**(*f_eval*, *t*, *y0*, *y_old*, *F*, *S*, *S_p*, *be_tol=1e-12*)

This runs SDC on the entire time step interval

$$\begin{cases} y^{[k+1]} - \Delta t\tilde{S}F(y^{[k+1]}) & = y_0 + \Delta t(S - \tilde{S})F(y^{[k]}) \\ \delta^{[k]} & = H(y^{[k]}) = y^{[k+1]} - y^{[k]} \end{cases}$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.

- **t** (*numpy.ndarray*) – the time nodes over a time step Δt
- **y0** (*numpy.ndarray*) – the initial solutions length, size of the problem
- **y_old** (*numpy.ndarray*) – the approximate solution $y^{[k]}$ [n_nodes x m]
- **F** (*numpy.ndarray*) – the derivative at the approximate solution $F(y^{[k]})$
- **S** (*numpy.ndarray*) – the spectral integration (Gaussian quadrature) matrix S
- **S_p** (*numpy.ndarray*) – the backward Euler integration matrix $\tilde{S}$
- **be_tol** (*float*) – the tolerance for the backward Euler solver

Returns the approximate solution, the correction

Return type numpy.ndarray (dimensions number of temporal nodes x size of the problem), numpy.ndarray (dimensions number of temporal nodes by size of the problem)

`integrator.`**`spectral`**(*f_eval*, *t*, *y0*, *S*, *f_tol=0.1*, *y_approx=None*, *do_print=False*, *verbose=False*)

This function solves directly the spectral solution (Gauss collocation formulation). That is, this function solves

$$y - \Delta t S F(y) = y_0$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*numpy.ndarray*) – the time nodes over a time step Δt
- **y0** (*numpy.ndarray*) – the initial solutions length, size of the problem
- **S** (*numpy.ndarray*) – the spectral integration matrix S
- **f_tol** (*float*) – the relative tolerance of the Newton-Krylov solver
- **y_approx** (*numpy.ndarray*) – the initial guess for the Newton-Krylov solver
- **verbose** (*bool*) – a flag for the built-in Newton-Krylov solver

Returns the spectral solution

Return type numpy.ndarray

`integrator.`**`spectral_root_finder`**(*f_eval*, *t*, *y*, *y0_vec*, *S*)

This function calculates the residual in the spectral (Gauss) collocation formulation. That is,

$$y - \Delta t S F(t, y) - y_0.$$

It is used in the root finding algorithm to solve

$$A(y) = y - \Delta t S F(t, y) - y_0 = 0$$

Parameters

- **f_eval** (*function*) – the derivative function $y' = f(t, y)$.
- **t** (*numpy.ndarray*) – the time nodes
- **y** (*numpy.ndarray*) – an approximate solution [number of temporal nodes x size of problem]
- **y0_vec** – the initial condition vector [number of temporal nodes x size of problem]
- **S** (*numpy.ndarray*) – the spectral integration (Gauss quadrature) matrix S

Returns the residual in the spectral collocation formulation

Return type numpy.ndarray [number of temporal nodes x size of problem]

`integrator.`**`step_size_scaler`**(*y*, *ysteps*, *n_nodes*, *k*, *tol*, *node_type*)

This function

Parameters

- **y** (*numpy.ndarray*) – an approximation of the ODE system using 1 step size of $h = \Delta t$
- **ysteps** (*numpy.ndarray*) – an approximation of the ODE system using k steps of size $h = \frac{\Delta t}{k}$
- **n_nodes** (*int*) – the number of temporal nodes within the time step
- **k** (*int*) – the amount of mini time steps leading up to $y(\Delta t)$
- **tol** (*float*) – the absolute error tolerance wanted within the time step
- **node_type** (*int*) – the type of temporal nodes

Given a step size Δt, calculate x the amount the step size should increase or decrease for the proper adaptive time step size Δt_{new} where

$$\Delta t_{new} = x\Delta t$$

Note: This function currently runs using only Gauss-Lobatto nodes

Returns

`integrator.`**`stopping_criteria`**(*t_final*, *t0*)

This function sends a flag whether or not we have reached the end of the simulation while taking account errors from inexact arithmetic.

Parameters

- **t_final** (*float*) – final time in the ODE simulation
- **t0** (*float*) – the start time for the current time step

Returns a flag indicating whether or not we have reached the end of the simulation

Return type bool

`integrator.`**`update_step_size`**(*dt*, *t0*, *t_final*)

This function makes sure that the time step Δt does not cause the simulation to go past the final time t_{final}.

Parameters

- **dt** (*float*) – the current step size Δt
- **t0** (*float*) – the start time for the current time step
- **t_final** (*float*) – the final time t_{final} of the ODE simulation

Returns the step size for the next time step

Return type float

1.3 my_globals module

This module contains some functions and constants that are useful for global use.

`my_globals.`**`load`**(*fname*)

This function loads data from a .pkl file.

Parameters **`fname`** (*`str`*) – the file name to be loaded from

Returns the data unpickled

`my_globals.`**`save`**(*x*, *fname*)

This function saves a python variable by pickling it.

Parameters

- **`x`** – the data to be saved
- **`fname`** (*`str`*) – the file name of the saved file. It must end with .pkl

1.4 params module

This module contains *`params.Params`* which contains information related to the differential equation system that is beings solved.

`class` `params.`**`Params`**(*n_nodes*, *m*, *node_type=2*)

Bases: `object`

This class includes information related to the differential equation system solver.

Parameters

- **`n_nodes`** (*`int`*) – the number of nodes in a time step
- **`m`** (*`int`*) – the size of the system
- **`node_type`** – the type of nodes used in the tine step

Variables

- **`n_nodes`** (*`int`*) – the number of temporal nodes
- **`m`** (*`int`*) – the size of the problem
- **`t`** (*`nunpy.ndarray`*) – the time nodes at such that $t \in [0, 1]$
- **`S`** (*`numpy.ndarray`*) – the spectral integration (Gauss quadrature) matrix
- **`S_p`** (*`numpy.ndarray`*) – the preconditioner integration matrix
- **`'spectral_radius'`** (*`float`*) – the spectral radius of the correction matrix from spectral deferred correction matrix.
- **`c`** (*`float`*) – the error constant in the error term from Gauss quadrature and Gauss-Lobatto nodes

1.5 plotter module

This module contains information about plotting.

`plotter.`**`corrections`**(*d_sdc_norm*, *d_jfnk_norm*, *y_spect_norm*, *do_rerr*, *labels=None*, *do_legend=False*)

This function plots the magnitude of the correction from the SDC method and JFNK method for each iteration.

Parameters

- **`d_sdc_norm`** (*`numpy.ndarray`*) – the maximum norm of the approximation $\|\delta_{sdc}^{[k]}\|$ from spectral deferred corrections (SDC) method
- **`d_jfnk_norm`** (*`numpy.ndarray`*) – the maximum norm of the approximation $\|\delta_{jfnk}^{[k]}\|$ from the Jacobian-Free Newton-Krylov (JFNK) method
- **`y_spect_norm`** (*`numpy.ndarray`*) – the norm of the spectral solution $\|y_{spect}\|$ used to scale the corrections for relative error
- **`do_rerr`** (*`bool`*) – the is flag indicates whether the relative error (if True) or the absolute error (if False) should be plotted
- **`labels`** (*`list`*) – these are the function
- **`do_legend`** (*`bool`*) – this flag indicates whether or not a legend will be shown

Returns None

`plotter.`**`plot_correction_time_steps`**(*D*, *Y*, *do_save=False*, *fpath=None*, *do_close=False*)

This function plots the magnitude of the corrections (both absolute and relative) $\|\delta\|$ vs iteration for ech time step.

Parameters

- **`D`** (*`list`*) – the corrections over the simulation. List of length number of time steps, containing a list of length number of iterations for the given time step, of the corrections for the iteration. The corrections are of dimensions (number of time nodes, size of the problem)
- **`Y`** (*`list`*) – the corresponding approximations to the given correction.
- **`do_save`** (*`bool`*) – indicating whether or not to save the data
- **`fpath`** (*`str`*) – the file path in which to save the data
- **`do_close`** (*`bool`*) – a flag indicating whether or not to close the plots

Returns None

`plotter.`**`plot_errors`**(*data*, *titles*, *labels*, *do_legend=False*, *main_title=”*, *xlabel=”*, *ylabel=”*, *ls='-'*)

This function plots the magnitude of the correction from the SDC method and JFNK method for each iteration.

Parameters

- **`data`** – errors from the simulations
- **`titles`** (*`list`*) – the titles of the subfigures
- **`labels`** (*`list`*) – the names of the lines
- **`do_legend`** (*`bool`*) – this flag indicates whether or not a legend will be shown
- **`main_title`** (*`str`*) – the title of the figure
- **`xlabel`** (*`str`*) – the x-axis label
- **`ylabel`** (*`str`*) – the y-axis label
- **`ls`** (*`str`*) – the line style

Returns None

1.6 points module

PyWENO quadrature points.

Requires SymPy. Namdi: I have edited this code from the original PyWENO code.

`points.`**`find_roots`**`(`*p*`)`

Return set of roots of polynomial *p*.

This uses the *nroots* method of the SymPy polynomial class to give rough roots, and subsequently refines these roots to arbitrary precision using mpmath.

Parameters **`p`** – sympy polynomial

Returns sorted *set* of roots.

`points.`**`gauss_legendre`**`(`*n*`)`

This function returns Gauss-Legendre nodes

Gauss-Legendre nodes are roots of $P_n(x)$.

Parameters **`n`** (*`int`*) – the number of nodes

Returns return Gauss-Legendre nodes $x \in [-1, 1]$

`points.`**`gauss_lobatto`**`(`*n*`)`

This function returns Gauss-Lobatto nodes. Gauss-Lobatto nodes are roots of $P'_{n-1}(x)$.

Parameters **`n`** (*`int`*) – the number of nodes

Returns Gauss-Lobatto nodes $x \in [-1, 1]$

`points.`**`gauss_radau`**`(`*n*`)`

Return Gauss-Radau nodes (right hand left hand point). Gauss-Radau nodes are roots of $P_n(x) + P_{n-1}(x)$.

Parameters **`n`** (*`int`*) – the number of nodes

Returns Gauss-Radau nodes $x \in [-1, 1]$

`points.`**`gauss_radau_2a`**`(`*n*`)`

Return Gauss-Radau 2a nodes (left hand left hand point). Gauss-Radau 2a nodes are roots of $P_n(x) - P_{n-1}(x)$.

Parameters **`n`** (*`int`*) – the number of nodes

Returns Gauss-Radau 2a nodes $x \in [-1, 1]$

`points.`**`legendre_poly`**`(`*n*`)`

Return Legendre polynomial $P_n(x)$.

Parameters **`n`** (*`int`*) – the degree of the polynomial

Returns Legendre polynomial

1.7 preprocess module

This module contains functions necessary for doing prepossessing before solving the differential equation system.

This module contains functions for the following

- temporal nodes in $t \in [0, 1]$
- backward Euler matrix
- spectral integration matrix

- spectral radius

`preprocess.`**`backward_euler_matrix`**`(`*t*`)`

This function calculates the backward Euler integration matrix

$\left(\tilde{S}\right)$ assuming $t \in [0, 1]$. That is, given $t = [t_0, t_1, \ldots, t_{n-1}]$ we have $0 \leq t_0 < t_1 < \ldots t_{n-1} \leq 1$.

$$\tilde{S} = \begin{bmatrix} \Delta t_0 & 0 & \cdots & 0 & 0 \\ \Delta t_0 & \Delta t_1 & \cdots & 0 & 0 \\ \cdot & \cdot & \cdots & 0 & 0 \\ \Delta t_0 & \Delta t_1 & \cdots & \Delta t_{n-3} & 0 \\ \Delta t_0 & \Delta t_1 & \cdots & \Delta t_{n-3} & \Delta t_{n-2} \end{bmatrix}$$

where

$$\begin{cases} \Delta t_0 & = t_0 - 0 \\ \Delta t_i & = t_i - t_{i-1} \; i = 1, \ldots, n-1 \end{cases}$$

Parameters **`t`** (*`np.ndarray`*) – the time nodes (length n)

Returns the backward Euler integration matrix $\tilde{S}$

Return type numpy.ndarray

`preprocess.`**`gauss_lobatto`**`(`*n*`)`

This function returns Gauss-Lobatto nodes $t \in [0, 1]$.

Parameters **`n`** – the number of nodes

Returns the Gauss-Lobatto nodes with $t \in [0, 1]$

Return type numpy.ndarray

`preprocess.`**`gauss_lobatto_error_constant`**`(`*n*`)`

This is the constant in the error term (R_n) for Gauss-Lobatto quadrature. That is,

$$\| \int_0^1 f(\tau)\,\mathrm{d}\tau - \sum_{j=0}^{n-1} w_j f(t_j) \| = R_n$$

where

$$\begin{aligned} R_n &= -\frac{n(n-1)^3[(n-2)!]^4}{(2n-1)[(2n-2)!]^3} \Delta t^{2n-1} f^{(2n-2)}(\tau), 0 < \tau < 1 \\ &= -c_n \Delta t^{2n-1} f^{(2n-2)}(\tau) \end{aligned}$$

Parameters **`n`** – the number of nodes

Returns the constant c in the error term R_n

`preprocess.`**`get_nodes`**`(`*n*, *node_type*`)`

This function various types of nodes $t \in [0, 1]$. Currently, this function may calculate the following types of nodes:

- Gauss-Legendre nodes
- Gauss-Lobatto nodes
- Guass Radau nodes (left end point)
- Gauss Radau 2a nodes (right end point)

Parameters

- **`n`** (*`int`*) – the number of nodes
- **`node_type`** (*`int`*) – the spectral node type

Returns the spectral nodes in $t \in [0, 1]$

`preprocess.`**`spectral_matrix`**`(`*t*`)`

This function calculates the spectral integration (Gaussian quadrature) matrix applied to the time step Δt where each entry is of S is given by

$$S_{ij} = \int_0^{t_i} \left(\prod_{k \neq j} \frac{t - t_k}{t_j - t_k} \right) \mathrm{d}t$$

Parameters **`t`** (*`numpy.ndarray`*) – the temporal nodes $t \in [0, 1]$. length n

Returns the spectral integration matrix

Return type numpy.ndarray

`preprocess.`**`spectral_radius`**`(`*node_type*, *S*, *S_p*`)`

This function calculates the spectral radius (the the largest magnitude of the eigenvalue) of the SDC correction matrix for the extremely stiff case.

First, calculate the correction matrix C for the extremely stiff case

$$\rho(C) = \rho(I - \tilde{S}^{-1} S)$$

Parameters

- **`node_type`** (*`int`*) – the type of node points
- **`S`** – the spectral integration matrix S
- **`S_p`** – the preconditioner integration matrix $\tilde{S}$

Returns the spectral radius

Return type float

CHAPTER 2

Examples Files

This directory contains the files of different ODE systems. They are the following:

1. stiff cosine system
2. the Van der Pol oscillator

2.1 cosine notebook

Let's see the performance of the numerical integrator for solving the following stiff, linear ODE system. This file solves the following ordinary differential equation (ODE) system:

$$\begin{aligned}\frac{dy_1(t)}{dt} &= \lambda_1(y_1(t) - \cos(t)) - \sin(t)\\ \frac{dy_2(t)}{dt} &= \lambda_2(y_2(t) - \cos(t)) - \sin(t)\\ \frac{dy_3(t)}{dt} &= \lambda_3(y_3(t) - \cos(t)) - \sin(t)\end{aligned}$$

where the initial condition is $y_1(0) = y_2(0) = y_3(0) = 1$. $\lambda_1, \lambda_2, \lambda_3$ are constants where $\lambda_i \leq 0$ and control the stiffness of the sytem.

The exact solution for this system is

$$y_1(t) = y_2(t) = y_3(t) = \cos(t)$$

Import

```
import sys
sys.path.append('..//source')

import matplotlib.pylab as plt
import numpy as np
import numpy.linalg as LA
```

```
from scipy.interpolate import CubicSpline
import scipy, time

import analysis, integrator, params, plotter

import points
```

```
%matplotlib notebook
```

```
# the stiffness parameter
K_stiff = -1/np.pi * np.array( (1e-3, 1e2, 1e5) )


def exact(t, m):

    n = len(t)
    y = np.zeros( (n, m) )

    for i in range(m):
        y[:,i] = np.cos(t)

    return y

def f_eval(t, y):

    """
    sovling
    dy_i/dt = k_i * (y_i - cos(t) ) - sin(t)

    :param float t: the time node
    :param numpy.ndarray: y the approximate solution length( m) at time node t
    """

    F = np.zeros( y.shape )
    m = len(y)

    # get the component-wise interaction
    for i in range(m):
        F[i] = K_stiff[i] * ( y[i] - np.cos(t) ) - np.sin(t)

    return F

def get_corrections_norm(D):

    """
    :param numpy.ndarray D: dimensions (the number of iterations, the number of nodes,
↪ the dimension of the problem)
    """

    y = np.vstack( [ np.linalg.norm(x, axis=0) for x in D] )

    return y

def get_stuff(p, dt):

    if p.t[-1] != 1:
        t = dt * np.hstack( [p.t, 1])
    else:
        t = dt * p.t

    S, S_p = dt * p.S, dt * p.S_p

    spectral_radius = p.spectral_radius
    node_type = p.node_type

    return t, S, S_p, spectral_radius, node_type
```

```
#
# set up integration parameters
#

# the parameters of the simulation
n_nodes      = 5

# the time step
#dt           = np.pi / 4
dt           = 1.0

# the dimension of the problem
m            = 3

# the initial solution
y0           = np.ones( (m,) )

# create the parameter object for Gauss-Lobatto nodes
p_lobatto = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_LOBATTO)

# create paramter for radau
#p_radau  = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_RADAU)

# create paramter for radau 2a
#p_radau_2a = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_RADAU_2A)

# legendre nodes
#p_legendre = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_LEGENDRE)

# backward euler, implicit tolerance. Also sused in SDC
BE_TOL = 1e-12

# SDC solver tolerance
SDC_TOL = 1e-12

# the maximum number of Newton iterations in the JFNK solver
N_ITER_MAX_NEWTON=integrator.N_ITER_MAX_NEWTON
```

Exact solution

```
y_exact = exact(dt * p_lobatto.t, m)
```

Backward Euler

```
nodes, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)
y_be = integrator.backward_euler(f_eval, nodes, y0, S_p, be_tol=BE_TOL, do_print=True)
```

```
t = nodes
LA.norm( y_be - y_exact, ord=np.inf)
```

```
elapsed time: 0.00[s]
```

```
0.11507996569196383
```

SDC

```
t, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)

# get the provisional solution
y_be = integrator.backward_euler(f_eval, t, y0, S_p, be_tol=BE_TOL)

# run SDC
y_sdc, Y_sdc, D_sdc, is_converged_sdc = integrator.sdc(f_eval, t, y0, y_be, S, S_p,␣
↪be_tol=BE_TOL, sdc_tol=SDC_TOL, do_print=True)
print('SDC converged: %s ' % is_converged_sdc)

# the norm of the corrections
d_norm_sdc, log_d_norm_sdc, d_norm_rel_sdc, log_d_norm_rel_sdc = analysis.analyze_
↪corrections(D_sdc, Y_sdc)
```

```
elapsed time: 0.09[s]
SDC converged: True
```

```
..//source/analysis.py:140: RuntimeWarning: divide by zero encountered in log10
  log_d_norm       = np.log10(d_norm)
..//source/analysis.py:141: RuntimeWarning: divide by zero encountered in log10
  log_d_norm_rel   = np.log10(d_norm_rel)
```

JFNK

```
t, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)

# get the provisional solution
y_be = integrator.backward_euler(f_eval, t, y0, S_p, be_tol=BE_TOL)

# run the JFNK
y_jfnk, Y_jfnk, D_jfnk, is_converged_jfnk, is_stiff,ratios = integrator.jfnk(f_eval,␣
↪t, y0, y_be, S, S_p, spectral_radius, be_tol=BE_TOL, \
                                                                             sdc_tol=SDC_
↪TOL, do_print=True)
print('JFNK converged: %s ' % is_converged_jfnk)

# the norm of the corrections
d_norm_jfnk, log_d_norm_jfnk, d_norm_rel_jfnk, log_d_norm_rel_jfnk = analysis.analyze_
↪corrections(D_jfnk, Y_jfnk)
```

```
elapsed time: 0.02[s]
JFNK converged: True
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
```

```
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  c, res, rank, s = np.linalg.lstsq(B, rhs)
..//source/analysis.py:140: RuntimeWarning: divide by zero encountered in␣
↪log10
  log_d_norm      = np.log10(d_norm)
..//source/analysis.py:141: RuntimeWarning: divide by zero encountered in␣
↪log10
  log_d_norm_rel  = np.log10(d_norm_rel)
```

Spectral solution

```
t, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)

# provisional solution
y_be = integrator.backward_euler(f_eval, t, y0, S_p, be_tol=BE_TOL)

# spectral solution
y_spect = integrator.run_spectral(f_eval, t, y0, S, tol=1e-10, y_approx=y_be,␣
↪verbose=True)
```

```
0:   |F(x)| = 1.67226e-05; step 1; tol 3.29637e-09
1:   |F(x)| = 2.44356e-10; step 1; tol 1.92168e-10
2:   |F(x)| = 1.38914e-12; step 1; tol 2.90862e-05
```

Plot the magnitude of deferred corrections (both SDC and JFNK corrections)

```
plt.figure()
plt.title('Magnitude of corrections ')

plt.plot( range(log_d_norm_sdc.shape[0]), log_d_norm_sdc.max(axis=1), '-o', label='sdc
↪')
plt.plot( range(log_d_norm_jfnk.shape[0]), log_d_norm_jfnk.max(axis=1), '-o', label=
↪'jfnk')

plt.xlabel('SDC iteration')
plt.ylabel('log10 (error)')
plt.legend(loc='best')

plt.show()
```

```
# final time
t_final = np.sqrt(4/3) * np.pi

# absolute error tolerance for tol
tol = 1e-11

# initial time step for the adaptive JFNK algorithm
dt_init = t_final / 10

print(t_final, dt_init)
```

```
3.6275987284684352 0.3627598728468435
```

Adaptive JFNK

```
t_adapt, y_adapt, Y_adapt, D_adapt, h_adapt = integrator.jfnk_adaptive(f_eval, t_
↪init=0, t_final=t_final, dt_init=dt_init, p=p_lobatto, y0=y0, \
                                                                  tol=tol, do_
↪print=True)
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  c, res, rank, s = np.linalg.lstsq(B, rhs)
```

```
elapsed time: 1.94[s]
```

Uniform JFNK

```
t_uni, y_uni, Y_uni, D_uni = integrator.jfnk_uniform(f_eval, t_init=0, t_final=t_
↪final, n_steps=12, p=p_lobatto, y0=y0, do_print=True)
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  c, res, rank, s = np.linalg.lstsq(B, rhs)
```

```
elapsed time: 0.27[s]
```

Reference solution

```
# the reference solution
n_steps_ref = 100
p_ref = params.Params(n_nodes=10, m=m, node_type=points.GAUSS_LOBATTO)

t_ref, y_ref, Y_ref, D_ref = integrator.jfnk_uniform(f_eval, t_init=0, t_final=t_
↪final, n_steps=n_steps_ref, p=p_ref, y0=y0, do_print=True)

# spline
y_spline = CubicSpline(t_ref, y_ref)

# errors
f = lambda t, y: ( LA.norm( y_spline(t) - y, ord=np.inf), integrator.relative_norm(y_
↪spline(t) - y, y_spline(t)) )
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  c, res, rank, s = np.linalg.lstsq(B, rhs)
```

```
elapsed time: 10.46[s]
```

Comparing the absolute and relative erros of the **uniform** solution to the reference solution

```
f(t_uni, y_uni)
```

```
(3.5825586941484744e-09, 1.5841824829072928e-09)
```

Comparing the absolute and relative errors of the **adaptive** solution to the reference solution

```
f(t_adapt, y_adapt)
```

```
(2.132772847218689e-10, 3.411275342907868e-11)
```

```
plotter.plot_correction_time_steps(D_uni, Y_uni)

plt.show()
```

Plot norm of corrections comparing

```
y_spect_norm = LA.norm(y_spect, axis=0)

labels=['y1', 'y2', 'y3']

plotter.corrections(d_norm_sdc, d_norm_jfnk, y_spect_norm, do_rerr=True,␣
↪labels=labels, do_legend=True)
plt.show()
```

```
..//source/plotter.py:150: RuntimeWarning: divide by zero encountered in log10
  temp = ax.plot(np.log10(y), ls, label=label)
/opt/local/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-
↪packages/matplotlib/legend.py:337: UserWarning: Automatic legend placement (loc=
↪"best") not implemented for figure legend. Falling back on "upper right".
  warnings.warn('Automatic legend placement (loc="best") not '
```

calculate absolute errors, relative errors

```
# absolute error, relative error

# error exact solution vs. spectral solution
aerr_spect, rerr_spect = analysis.error_analysis(y_spect, y_exact)

# spectral solution vs. backward euler solution
aerr_be, rerr_be = analysis.error_analysis(y_be, y_spect)

# spectral solution vs. SDC solution
aerr_sdc, rerr_sdc = analysis.error_analysis(y_sdc, y_spect)

# spectral solution vs. JFNK solution
aerr_jfnk, rerr_jfnk = analysis.error_analysis(y_jfnk, y_spect)
```

2.2 van_der_pol notebook

Let's now examine the performance of integration methods for the following stiff, non-linear ODE system, **Van der Pol's oscillator**. This file solves This file solves the following ordinary differential equation (ODE) system:

$$\frac{dy_1(t)}{dt} = y_2(t)$$
$$\frac{dy_2(t)}{dt} = \lambda\left[1 - y_1^2(t)\right] y_2(t) - y_1(t)$$

where $\lambda \geq 0$ and $[y_1(0), y_2(0)]^T = [2, 1]^T$.

Import

```
import sys
sys.path.append('..//source')

import matplotlib.pylab as plt
import numpy as np
import numpy.linalg as LA

import scipy, time
from scipy.interpolate import CubicSpline

import analysis, integrator, params, plotter

import points
```

```
%matplotlib notebook
```

Functions

```
# the stiffness parameter
K_stiff = 20
# https://www.cs.ox.ac.uk/files/1175/lecture06-handout.pdf

# backward euler, implicit tolerance. Also sused in SDC
BE_TOL = 1e-12
SDC_TOL = 1e-18 #1e-12
N_ITER_MAX_NEWTON=50

def f_eval(t, y):

    """
    sovling
    dy1/dt = y2
    dy2/dt = k * (1 - y1^2)y2 - y1

    :param float t: the time node
    :param numpy.ndarray: y the approximate solution length( m) at time node t
    """

    F = np.zeros( y.shape )
    m = len(y)

    # get the component-wise interaction
    F[0] = y[1]
    F[1] = K_stiff * (1 - y[0]**2) * y[1] - y[0]
    return F

def get_corrections_norm(D):

    """
    :param numpy.ndarray D: dimensions (the number of iterations, the number of nodes,
↪ the dimension of the problem)
    """

    y = np.vstack( [ np.linalg.norm(x, axis=0) for x in D] )

    return y

def get_stuff(p, dt):

    if p.t[-1] != 1:
        t = dt * np.hstack( [p.t, 1])
    else:
        t = dt * p.t

    S, S_p = dt * p.S, dt * p.S_p

    spectral_radius = p.spectral_radius
    node_type = p.node_type

    return t, S, S_p, spectral_radius, node_type
```

Parameters

```
#
# set up integration parameters
#

# the parameters of the simulation
n_nodes     = 10

# the time step
dt = .25

# the dimension of the problem
m           = 2

# the initial solution
y0 = np.array( (2, 1) )

# create the parameter object
p_lobatto = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_LOBATTO)

# create paramter for radau
#p_radau  = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_RADAU)

# create paramter for radau 2a
#p_radau_2a = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_RADAU_2A)

# legendre nodes
#p_legendre = params.Params(n_nodes=n_nodes, m=m, node_type=points.GAUSS_LEGENDRE)
```

Backward Euler

```
nodes, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)
y_be = integrator.backward_euler(f_eval, nodes, y0, S_p, be_tol=BE_TOL, do_print=True)

t = nodes
#LA.norm( y_be - exact(t), ord=np.inf)
```

```
elapsed time: 0.00[s]
```

SDC

```
t, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)

y_be = integrator.backward_euler(f_eval, t, y0, S_p, be_tol=BE_TOL)

y_sdc, Y_sdc, D_sdc, is_converged_sdc = integrator.sdc(f_eval, t, y0, y_be, S, S_p,␣
↪be_tol=BE_TOL, sdc_tol=SDC_TOL, do_print=True)
print('SDC converged: %s ' % is_converged_sdc)

# the norm of the corrections
d_norm_sdc, log_d_norm_sdc, d_norm_rel_sdc, log_d_norm_rel_sdc = analysis.analyze_
↪corrections(D_sdc, Y_sdc)
```

```
elapsed time: 0.09[s]
SDC converged: True
```

```
..//source/analysis.py:140: RuntimeWarning: divide by zero encountered in log10
  log_d_norm      = np.log10(d_norm)
..//source/analysis.py:141: RuntimeWarning: divide by zero encountered in log10
  log_d_norm_rel  = np.log10(d_norm_rel)
```

JFNK

```
t, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)

y_be = integrator.backward_euler(f_eval, t, y0, S_p, be_tol=BE_TOL)

y_jfnk, Y_jfnk, D_jfnk, is_converged_jfnk, is_stiff,ratios = integrator.jfnk(f_eval,␣
↪t, y0, y_be, S, S_p, spectral_radius, be_tol=BE_TOL, \
                                                                       sdc_tol=SDC_
↪TOL, do_print=True)
print('JFNK converged: %s ' % is_converged_jfnk)

# the norm of the corrections
d_norm_jfnk, log_d_norm_jfnk, d_norm_rel_jfnk, log_d_norm_rel_jfnk = analysis.analyze_
↪corrections(D_jfnk, Y_jfnk)
```

```
elapsed time: 0.06[s]
JFNK converged: True
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  c, res, rank, s = np.linalg.lstsq(B, rhs)
```

```
..//source/analysis.py:140: RuntimeWarning: divide by zero encountered in␣
↪log10
  log_d_norm      = np.log10(d_norm)
..//source/analysis.py:141: RuntimeWarning: divide by zero encountered in␣
↪log10
  log_d_norm_rel  = np.log10(d_norm_rel)
```

Spectral Solution

```
t, S, S_p, spectral_radius, node_type = get_stuff(p_lobatto, dt)

y_be = integrator.backward_euler(f_eval, t, y0, S_p, be_tol=BE_TOL)

y_spect = integrator.run_spectral(f_eval, t, y0, S, tol=1e-10, y_approx=y_be,␣
↪verbose=True)
```

```
0:  |F(x)| = 0.00299999; step 1; tol 3.94638e-06
1:  |F(x)| = 8.14323e-08; step 1; tol 6.63128e-10
2:  |F(x)| = 7.61131e-16; step 1; tol 7.86262e-17
```

Plot the corrections

```
plt.figure()
plt.title('Magnitude of corrections ')

plt.plot( range(log_d_norm_sdc.shape[0]), log_d_norm_sdc.max(axis=1), '-o', label='sdc
↪')
plt.plot( range(log_d_norm_jfnk.shape[0]), log_d_norm_jfnk.max(axis=1), '-o', label=
↪'jfnk')

plt.xlabel('SDC iteration')
plt.ylabel('log10 (error)')
plt.legend(loc='best')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
t_final =1
tol = 1e-5
dt_init = 4e-1

print(t_final, dt_init)
```

```
1 0.4
```

Adaptive JFNK

```
t_adapt, y_adapt, Y_adapt, D_adapt, h_adapt = integrator.jfnk_adaptive(f_eval, t_
↪init=0, t_final=t_final, dt_init=dt_init, p=p_lobatto, y0=y0, \
                                                                      tol=tol, do_
↪print=True)
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
```

```
  c, res, rank, s = np.linalg.lstsq(B, rhs)
```

```
elapsed time: 0.35[s]
```

```
# the time steps
h_adapt
```

```
array([0.3190401 , 0.3190401 , 0.36191979])
```

Uniform JFNK

```
t_uni, y_uni, Y_uni, D_uni = integrator.jfnk_uniform(f_eval, t_init=0, t_final=t_
↪final, n_steps=12, p=p_lobatto, y0=y0, do_print=True)
```

```
..//source/integrator.py:721: FutureWarning: rcond parameter will change␣
↪to the default of machine precision times max(M, N) where M and N are the␣
↪input matrix dimensions.
To use the future default and silence this warning we advise to pass␣
↪rcond=None, to keep using the old, explicitly pass rcond=-1.
  c, res, rank, s = np.linalg.lstsq(B, rhs)
```

```
elapsed time: 0.24[s]
```

Reference solution

```
# the reference solution
n_steps_ref = 100
p_ref = params.Params(n_nodes=10, m=m, node_type=points.GAUSS_LOBATTO)

t_ref, y_ref, Y_ref, D_ref = integrator.jfnk_uniform(f_eval, t_init=0, t_final=t_
↪final, n_steps=n_steps_ref, p=p_ref, y0=y0, do_print=True)

# spline
y_spline = CubicSpline(t_ref, y_ref)

# errors
f = lambda t, y: ( LA.norm( y_spline(t) - y, ord=np.inf), integrator.relative_norm(y_
↪spline(t) - y, y_spline(t)) )
```

```
elapsed time: 1.01[s]
```

Comparing the absolute and relative erros of the **uniform** solution to the reference solution

```
f(t_uni, y_uni)
```

```
(7.490298863967182e-07, 1.1106575637047464e-06)
```

Comparing the absolute and relative errors of the **adaptive** solution to the reference solution

```
f(t_adapt, y_adapt), tol
```

```
((0.0016080388293931436, 0.0031861992483273984), 1e-05)
```

Plot the JFNK corrections for each time step

```
plotter.plot_correction_time_steps(D_uni, Y_uni)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

Plot norm of corrections

```
y_spect_norm = LA.norm(y_spect, axis=0)

labels=['y1', 'y2']

plotter.corrections(d_norm_sdc, d_norm_jfnk, y_spect_norm, do_rerr=True,␣
↪labels=labels, do_legend=True)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
..//source/plotter.py:150: RuntimeWarning: divide by zero encountered in log10
  temp = ax.plot(np.log10(y), ls, label=label)
/Users/nmadi/anaconda/lib/python3.5/site-packages/matplotlib/legend.py:652:␣
↪UserWarning: Automatic legend placement (loc="best") not implemented for figure␣
↪legend. Falling back on "upper right".
  warnings.warn('Automatic legend placement (loc="best") not '
```

```
#
# error
#
aerr_be, rerr_be = analysis.error_analysis(y_be, y_spect)
aerr_sdc, rerr_sdc = analysis.error_analysis(y_sdc, y_spect)
aerr_jfnk, rerr_jfnk = analysis.error_analysis(y_jfnk, y_spect)
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

References

Bibliography

[QBC+16] Wenzhen Qu, Namdi Brandon, Dangxing Chen, Jingfang Huang, and Tyler Kress. A numerical framework for integrating deferred correction methods to solve high order collocation formulations of odes. *Journal of Scientific Computing*, 68(2):484–520, Aug 2016. URL: https://doi.org/10.1007/s10915-015-0146-9, doi:10.1007/s10915-015-0146-9.

Python Module Index