Using the Stan Math C++ Library

Stan Development Team

2024-07-15

Using the StanHeaders Package from Other R Packages

The **StanHeaders** package contains no R functions. To use the Stan Math Library in other packages, it is often sufficient to specify

```
LinkingTo: StanHeaders (>= 2.26.0), RcppParallel (>= 5.0.1)
```

in the DESCRIPTION file of another package and put something like

```
CXX_STD = CXX17
PKG_CXXFLAGS = $(shell "$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" -e "RcppParallel::CxxFlags()") \
               $(shell "$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" -e "StanHeaders:::CxxFlags()")
PKG_LIBS = $(shell "$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" -e "RcppParallel::RcppParallelLibs()")
\
           $(shell "$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" -e "StanHeaders:::LdFlags()")
```

in the src/Makevars and src/Makevars.win files and put `GNU make` in the `SystemRequirements:` field of the package's DESCRIPTION file. If, in addition, the other package needs to utilize the MCMC, optimization, variational inference, or parsing facilities of the Stan Library, then it is also necessary to include the `src` directory of **StanHeaders** in the other package's `PKG_CXXFLAGS` in the src/Makevars and src/Makevars.win files with something like

```
STANHEADERS_SRC = $(shell "$(R_HOME)/bin$(R_ARCH_BIN)/Rscript" -e "message()" \
  -e "cat(system.file('include', 'src', package = 'StanHeaders', mustWork = TRUE))" \
```

```
  -e "message()" | grep "StanHeaders")
PKG_CXXFLAGS += -I"$(STANHEADERS_SRC)"
```

Calling functions in the StanHeaders Package from R

The only exposed R function in the in the **StanHeaders** package is `stanFunction`, which can be used to call most functions in the Stan Math Library.

```
example(stanFunction, package = "StanHeaders", run.dontrun = TRUE)
#>
#> stnFnc>   files <- dir(system.file("include", "stan", "math", "prim",
#> stnFnc+                              package = "StanHeaders"),
#> stnFnc+                pattern = "hpp$", recursive = TRUE)
#>
#> stnFnc>   functions <- sub("\\.hpp$", "",
#> stnFnc+                    sort(unique(basename(files[dirname(files) != "."]))))
#>
#> stnFnc>   length(functions) # you could call most of these Stan functions
#> [1] 955
#>
#> stnFnc>     log(sum(exp(exp(1)), exp(pi))) # true value
#> [1] 3.645318
#>
#> stnFnc>     stanFunction("log_sum_exp", x = exp(1), y = pi)
#> [1] 3.645318
#>
#> stnFnc>     args(log_sum_exp) # now exists in .GlobalEnv
#> function (x, y)
#> NULL
#>
#> stnFnc>     log_sum_exp(x = pi, y = exp(1))
#> [1] 3.645318
#>
#> stnFnc>     # but log_sum_exp() was not defined for a vector or matrix
```

```
#> stnFnc>     x <- c(exp(1), pi)
#>
#> stnFnc>     try(log_sum_exp(x))
#> Error in log_sum_exp(x) : argument "y" is missing, with no default
#>
#> stnFnc>     stanFunction("log_sum_exp", x = x) # now it is
#> [1] 3.645318
#>
#> stnFnc>     # log_sum_exp() is now also defined for a matrix
#> stnFnc>     log_sum_exp(as.matrix(x))
#> [1] 3.645318
#>
#> stnFnc>     log_sum_exp(t(as.matrix(x)))
#> [1] 3.645318
#>
#> stnFnc>     log_sum_exp(rbind(x, x))
#> [1] 4.338465
#>
#> stnFnc>     # but log_sum_exp() was not defined for a list
#> stnFnc>     try(log_sum_exp(as.list(x)))
#> Error in eval(ei, envir) :
#>   Not compatible with requested type: [type=list; target=double].
#>
#> stnFnc>     stanFunction("log_sum_exp", x = as.list(x)) # now it is
#> [1] 3.645318
#>
#> stnFnc>     # in rare cases, passing a nested list is needed
#> stnFnc>     stanFunction("dims", x = list(list(1:3)))
#> [1] 1 1 3
#>
#> stnFnc>     # functions of complex arguments work
#> stnFnc>     stanFunction("eigenvalues", # different ordering than base:eigen()
#> stnFnc+                  x = matrix(complex(real = 1:9, imaginary = pi),
#> stnFnc+                             nrow = 3, ncol = 3))
#> [1] -8.179880e-17+1.622085e-18i -8.612657e-01+4.854073e-01i  1.586127e+01+8.939371e+00i
#>
#> stnFnc>     # nullary functions work but are not that interesting
#> stnFnc>     stanFunction("negative_infinity")
```

```
#> [1] -Inf
#>
#> stnFnc>     # PRNG functions work by adding a seed argument
#> stnFnc>     stanFunction("lkj_corr_rng", K = 3L, eta = 1)
#>            [,1]      [,2]       [,3]
#> [1,] 1.00000000 0.5654780 0.03415387
#> [2,] 0.56547796 1.0000000 0.24813894
#> [3,] 0.03415387 0.2481389 1.00000000
#>
#> stnFnc>     args(lkj_corr_rng) # has a seed argument
#> function (K, eta, random_seed = sample.int(.Machine$integer.max,
#>     size = 1L))
#> NULL
```

The `functions` object defined in this example lists the many Stan functions that could be called (if all their arguments are numeric, see `help(stanFunction, package = "StanHeaders")` for details)

```
#>        [,1]                              [,2]
#>   [1,] "Eigen"                           "LDLT_factor"
#>   [2,] "Phi"                             "Phi_approx"
#>   [3,] "StdVectorBuilder"                "VectorBuilder"
#>   [4,] "VectorBuilderHelper"             "abs"
#>   [5,] "accumulator"                     "acos"
#>   [6,] "acosh"                           "ad_promotable"
#>   [7,] "add"                             "add_diag"
#>   [8,] "algebra_solver_adapter"          "all"
#>   [9,] "any"                             "append_array"
#>  [10,] "append_col"                      "append_return_type"
#>  [11,] "append_row"                      "apply"
#>  [12,] "apply_scalar_binary"             "apply_scalar_ternary"
#>  [13,] "apply_scalar_unary"              "apply_vector_unary"
#>  [14,] "arg"                             "array_builder"
#>  [15,] "as_array_or_scalar"              "as_bool"
#>  [16,] "as_column_vector_or_scalar"      "as_value_array_or_scalar"
#>  [17,] "as_value_column_array_or_scalar" "as_value_column_vector_or_scalar"
#>  [18,] "asin"                            "asinh"
#>  [19,] "assign"                          "atan"
```

```
#>  [20,] "atan2"                      "atanh"
#>  [21,] "autocorrelation"            "autocovariance"
#>  [22,] "base_type"                  "bernoulli_ccdf_log"
#>  [23,] "bernoulli_cdf"              "bernoulli_cdf_log"
#>  [24,] "bernoulli_lccdf"            "bernoulli_lcdf"
#>  [25,] "bernoulli_log"              "bernoulli_logit_glm_log"
#>  [26,] "bernoulli_logit_glm_lpmf"   "bernoulli_logit_glm_rng"
#>  [27,] "bernoulli_logit_log"        "bernoulli_logit_lpmf"
#>  [28,] "bernoulli_logit_rng"        "bernoulli_lpmf"
#>  [29,] "bernoulli_rng"              "bessel_first_kind"
#>  [30,] "bessel_second_kind"         "beta"
#>  [31,] "beta_binomial_ccdf_log"     "beta_binomial_cdf"
#>  [32,] "beta_binomial_cdf_log"      "beta_binomial_lccdf"
#>  [33,] "beta_binomial_lcdf"         "beta_binomial_log"
#>  [34,] "beta_binomial_lpmf"         "beta_binomial_rng"
#>  [35,] "beta_ccdf_log"              "beta_cdf"
#>  [36,] "beta_cdf_log"               "beta_lccdf"
#>  [37,] "beta_lcdf"                  "beta_log"
#>  [38,] "beta_lpdf"                  "beta_proportion_ccdf_log"
#>  [39,] "beta_proportion_cdf_log"    "beta_proportion_lccdf"
#>  [40,] "beta_proportion_lcdf"       "beta_proportion_log"
#>  [41,] "beta_proportion_lpdf"       "beta_proportion_rng"
#>  [42,] "beta_rng"                   "binary_log_loss"
#>  [43,] "binomial_ccdf_log"          "binomial_cdf"
#>  [44,] "binomial_cdf_log"           "binomial_coefficient_log"
#>  [45,] "binomial_lccdf"             "binomial_lcdf"
#>  [46,] "binomial_log"               "binomial_logit_log"
#>  [47,] "binomial_logit_lpmf"        "binomial_lpmf"
#>  [48,] "binomial_rng"               "block"
#>  [49,] "bool_constant"              "boost_policy"
#>  [50,] "broadcast_array"            "categorical_log"
#>  [51,] "categorical_logit_glm_lpmf" "categorical_logit_log"
#>  [52,] "categorical_logit_lpmf"     "categorical_logit_rng"
#>  [53,] "categorical_lpmf"           "categorical_rng"
#>  [54,] "cauchy_ccdf_log"            "cauchy_cdf"
#>  [55,] "cauchy_cdf_log"             "cauchy_lccdf"
#>  [56,] "cauchy_lcdf"                "cauchy_log"
#>  [57,] "cauchy_lpdf"                "cauchy_rng"
```

```
#>  [58,] "cbrt"                        "ceil"
#>  [59,] "check_2F1_converges"         "check_3F2_converges"
#>  [60,] "check_bounded"               "check_cholesky_factor"
#>  [61,] "check_cholesky_factor_corr"  "check_column_index"
#>  [62,] "check_consistent_size"       "check_consistent_sizes"
#>  [63,] "check_consistent_sizes_mvt"  "check_corr_matrix"
#>  [64,] "check_cov_matrix"            "check_finite"
#>  [65,] "check_flag_sundials"         "check_greater"
#>  [66,] "check_greater_or_equal"      "check_ldlt_factor"
#>  [67,] "check_less"                  "check_less_or_equal"
#>  [68,] "check_lower_triangular"      "check_matching_dims"
#>  [69,] "check_matching_sizes"        "check_multiplicable"
#>  [70,] "check_nonnegative"           "check_nonzero_size"
#>  [71,] "check_not_nan"               "check_ordered"
#>  [72,] "check_pos_definite"          "check_pos_semidefinite"
#>  [73,] "check_positive"              "check_positive_finite"
#>  [74,] "check_positive_ordered"      "check_range"
#>  [75,] "check_row_index"             "check_simplex"
#>  [76,] "check_size_match"            "check_sorted"
#>  [77,] "check_square"                "check_std_vector_index"
#>  [78,] "check_symmetric"             "check_unit_vector"
#>  [79,] "check_vector"                "check_vector_index"
#>  [80,] "chi_square_ccdf_log"         "chi_square_cdf"
#>  [81,] "chi_square_cdf_log"          "chi_square_lccdf"
#>  [82,] "chi_square_lcdf"             "chi_square_log"
#>  [83,] "chi_square_lpdf"             "chi_square_rng"
#>  [84,] "child_type"                  "chol2inv"
#>  [85,] "cholesky_corr_constrain"     "cholesky_corr_free"
#>  [86,] "cholesky_decompose"          "cholesky_factor_constrain"
#>  [87,] "cholesky_factor_free"        "choose"
#>  [88,] "col"                         "cols"
#>  [89,] "columns_dot_product"         "columns_dot_self"
#>  [90,] "compiler_attributes"         "complex_base"
#>  [91,] "complex_schur_decompose"     "conj"
#>  [92,] "conjunction"                 "constants"
#>  [93,] "constraint_tolerance"        "contains_fvar"
#>  [94,] "contains_std_vector"         "copysign"
#>  [95,] "corr_constrain"              "corr_free"
```

```
#>  [96,] "corr_matrix_constrain"          "corr_matrix_free"
#>  [97,] "cos"                            "cosh"
#>  [98,] "coupled_ode_system"             "cov_exp_quad"
#>  [99,] "cov_matrix_constrain"           "cov_matrix_constrain_lkj"
#> [100,] "cov_matrix_free"                "cov_matrix_free_lkj"
#> [101,] "crossprod"                      "csr_extract"
#> [102,] "csr_extract_u"                  "csr_extract_v"
#> [103,] "csr_extract_w"                  "csr_matrix_times_vector"
#> [104,] "csr_to_dense_matrix"            "csr_u_to_z"
#> [105,] "cumulative_sum"                 "determinant"
#> [106,] "diag_matrix"                    "diag_post_multiply"
#> [107,] "diag_pre_multiply"              "diagonal"
#> [108,] "digamma"                        "dims"
#> [109,] "dirichlet_log"                  "dirichlet_lpdf"
#> [110,] "dirichlet_lpmf"                 "dirichlet_rng"
#> [111,] "discrete_range_ccdf_log"        "discrete_range_cdf"
#> [112,] "discrete_range_cdf_log"         "discrete_range_lccdf"
#> [113,] "discrete_range_lcdf"            "discrete_range_log"
#> [114,] "discrete_range_lpmf"            "discrete_range_rng"
#> [115,] "disjunction"                    "distance"
#> [116,] "divide"                         "divide_columns"
#> [117,] "domain_error"                   "domain_error_vec"
#> [118,] "dot"                            "dot_product"
#> [119,] "dot_self"                       "double_exponential_ccdf_log"
#> [120,] "double_exponential_cdf"         "double_exponential_cdf_log"
#> [121,] "double_exponential_lccdf"       "double_exponential_lcdf"
#> [122,] "double_exponential_log"         "double_exponential_lpdf"
#> [123,] "double_exponential_rng"         "eigen_comparisons"
#> [124,] "eigendecompose"                 "eigendecompose_sym"
#> [125,] "eigenvalues"                    "eigenvalues_sym"
#> [126,] "eigenvectors"                   "eigenvectors_sym"
#> [127,] "elementwise_check"              "elt_divide"
#> [128,] "elt_multiply"                   "erf"
#> [129,] "erfc"                           "error_index"
#> [130,] "eval"                           "exp"
#> [131,] "exp2"                           "exp_mod_normal_ccdf_log"
#> [132,] "exp_mod_normal_cdf"             "exp_mod_normal_cdf_log"
#> [133,] "exp_mod_normal_lccdf"           "exp_mod_normal_lcdf"
```

```
#> [134,] "exp_mod_normal_log"              "exp_mod_normal_lpdf"
#> [135,] "exp_mod_normal_rng"              "expm1"
#> [136,] "exponential_ccdf_log"            "exponential_cdf"
#> [137,] "exponential_cdf_log"             "exponential_lccdf"
#> [138,] "exponential_lcdf"                "exponential_log"
#> [139,] "exponential_lpdf"                "exponential_rng"
#> [140,] "fabs"                            "factor_U"
#> [141,] "factor_cov_matrix"               "falling_factorial"
#> [142,] "fdim"                            "fft"
#> [143,] "fill"                            "finite_diff_gradient"
#> [144,] "finite_diff_gradient_auto"       "finite_diff_stepsize"
#> [145,] "floor"                           "fma"
#> [146,] "fmax"                            "fmin"
#> [147,] "fmod"                            "for_each"
#> [148,] "forward_as"                      "frechet_ccdf_log"
#> [149,] "frechet_cdf"                     "frechet_cdf_log"
#> [150,] "frechet_lccdf"                   "frechet_lcdf"
#> [151,] "frechet_log"                     "frechet_lpdf"
#> [152,] "frechet_rng"                     "gamma_ccdf_log"
#> [153,] "gamma_cdf"                       "gamma_cdf_log"
#> [154,] "gamma_lccdf"                     "gamma_lcdf"
#> [155,] "gamma_log"                       "gamma_lpdf"
#> [156,] "gamma_p"                         "gamma_q"
#> [157,] "gamma_rng"                       "gaussian_dlm_obs_log"
#> [158,] "gaussian_dlm_obs_lpdf"           "gaussian_dlm_obs_rng"
#> [159,] "generalized_inverse"             "get"
#> [160,] "get_base1"                       "get_base1_lhs"
#> [161,] "get_imag"                        "get_lp"
#> [162,] "get_real"                        "gp_dot_prod_cov"
#> [163,] "gp_exp_quad_cov"                 "gp_exponential_cov"
#> [164,] "gp_matern32_cov"                 "gp_matern52_cov"
#> [165,] "gp_periodic_cov"                 "grad_2F1"
#> [166,] "grad_F32"                        "grad_inc_beta"
#> [167,] "grad_pFq"                        "grad_reg_inc_beta"
#> [168,] "grad_reg_inc_gamma"              "grad_reg_lower_inc_gamma"
#> [169,] "gumbel_ccdf_log"                 "gumbel_cdf"
#> [170,] "gumbel_cdf_log"                  "gumbel_lccdf"
#> [171,] "gumbel_lcdf"                     "gumbel_log"
```

```
#> [172,] "gumbel_lpdf"                  "gumbel_rng"
#> [173,] "hcubature"                    "head"
#> [174,] "hmm_check"                    "hmm_hidden_state_prob"
#> [175,] "hmm_latent_rng"               "hmm_marginal"
#> [176,] "holder"                       "hypergeometric_2F1"
#> [177,] "hypergeometric_2F2"           "hypergeometric_3F2"
#> [178,] "hypergeometric_log"           "hypergeometric_lpmf"
#> [179,] "hypergeometric_pFq"           "hypergeometric_rng"
#> [180,] "hypot"                        "i_times"
#> [181,] "identity_constrain"           "identity_free"
#> [182,] "identity_matrix"              "if_else"
#> [183,] "imag"                         "inc_beta"
#> [184,] "inc_beta_dda"                 "inc_beta_ddb"
#> [185,] "inc_beta_ddz"                 "include_summand"
#> [186,] "index_apply"                  "index_type"
#> [187,] "init_threadpool_tbb"          "initialize"
#> [188,] "initialize_fill"              "int_step"
#> [189,] "integrate_1d"                 "integrate_1d_adapter"
#> [190,] "integrate_ode_rk45"           "integrate_ode_std_vector_interface_adapter"
#> [191,] "inv"                          "inv_Phi"
#> [192,] "inv_chi_square_ccdf_log"      "inv_chi_square_cdf"
#> [193,] "inv_chi_square_cdf_log"       "inv_chi_square_lccdf"
#> [194,] "inv_chi_square_lcdf"          "inv_chi_square_log"
#> [195,] "inv_chi_square_lpdf"          "inv_chi_square_rng"
#> [196,] "inv_cloglog"                  "inv_erfc"
#> [197,] "inv_gamma_ccdf_log"           "inv_gamma_cdf"
#> [198,] "inv_gamma_cdf_log"            "inv_gamma_lccdf"
#> [199,] "inv_gamma_lcdf"               "inv_gamma_log"
#> [200,] "inv_gamma_lpdf"               "inv_gamma_rng"
#> [201,] "inv_inc_beta"                 "inv_logit"
#> [202,] "inv_sqrt"                     "inv_square"
#> [203,] "inv_wishart_cholesky_lpdf"    "inv_wishart_cholesky_rng"
#> [204,] "inv_wishart_log"              "inv_wishart_lpdf"
#> [205,] "inv_wishart_rng"              "invalid_argument"
#> [206,] "invalid_argument_vec"         "inverse"
#> [207,] "inverse_softmax"              "inverse_spd"
#> [208,] "is_any_nan"                   "is_arena_matrix"
#> [209,] "is_autodiff"                  "is_base_pointer_convertible"
```

```
#> [210,] "is_cholesky_factor"              "is_cholesky_factor_corr"
#> [211,] "is_column_index"                 "is_complex"
#> [212,] "is_constant"                     "is_container"
#> [213,] "is_container_or_var_matrix"      "is_corr_matrix"
#> [214,] "is_dense_dynamic"                "is_detected"
#> [215,] "is_double_or_int"                "is_eigen"
#> [216,] "is_eigen_dense_base"             "is_eigen_dense_dynamic"
#> [217,] "is_eigen_matrix"                 "is_eigen_matrix_base"
#> [218,] "is_eigen_sparse_base"            "is_fvar"
#> [219,] "is_inf"                          "is_integer"
#> [220,] "is_kernel_expression"            "is_ldlt_factor"
#> [221,] "is_less_or_equal"                "is_lower_triangular"
#> [222,] "is_mat_finite"                   "is_matching_dims"
#> [223,] "is_matching_size"                "is_matrix"
#> [224,] "is_matrix_cl"                    "is_nan"
#> [225,] "is_nonpositive_integer"          "is_nonzero_size"
#> [226,] "is_not_nan"                      "is_ordered"
#> [227,] "is_plain_type"                   "is_pos_definite"
#> [228,] "is_positive"                     "is_rev_matrix"
#> [229,] "is_scal_finite"                  "is_size_match"
#> [230,] "is_square"                       "is_stan_scalar"
#> [231,] "is_stan_scalar_or_eigen"         "is_string_convertible"
#> [232,] "is_symmetric"                    "is_tuple"
#> [233,] "is_uninitialized"                "is_unit_vector"
#> [234,] "is_var"                          "is_var_and_matrix_types"
#> [235,] "is_var_dense_dynamic"            "is_var_eigen"
#> [236,] "is_var_matrix"                   "is_var_or_arithmetic"
#> [237,] "is_vari"                         "is_vector"
#> [238,] "is_vector_like"                  "isfinite"
#> [239,] "isinf"                           "isnan"
#> [240,] "isnormal"                        "lambert_w"
#> [241,] "lb_constrain"                    "lb_free"
#> [242,] "lbeta"                           "ldexp"
#> [243,] "lgamma"                          "lgamma_stirling"
#> [244,] "lgamma_stirling_diff"            "linspaced_array"
#> [245,] "linspaced_int_array"             "linspaced_row_vector"
#> [246,] "linspaced_vector"                "lkj_corr_cholesky_log"
#> [247,] "lkj_corr_cholesky_lpdf"          "lkj_corr_cholesky_rng"
```

```
#> [248,] "lkj_corr_log"                "lkj_corr_lpdf"
#> [249,] "lkj_corr_rng"                "lkj_cov_log"
#> [250,] "lkj_cov_lpdf"                "lmgamma"
#> [251,] "lmultiply"                   "log"
#> [252,] "log10"                       "log1m"
#> [253,] "log1m_exp"                   "log1m_inv_logit"
#> [254,] "log1p"                       "log1p_exp"
#> [255,] "log2"                        "log_determinant"
#> [256,] "log_determinant_ldlt"        "log_determinant_spd"
#> [257,] "log_diff_exp"                "log_falling_factorial"
#> [258,] "log_inv_logit"               "log_inv_logit_diff"
#> [259,] "log_mix"                     "log_modified_bessel_first_kind"
#> [260,] "log_rising_factorial"        "log_softmax"
#> [261,] "log_sum_exp"                 "log_sum_exp_signed"
#> [262,] "logb"                        "logical_and"
#> [263,] "logical_eq"                  "logical_gt"
#> [264,] "logical_gte"                 "logical_lt"
#> [265,] "logical_lte"                 "logical_negation"
#> [266,] "logical_neq"                 "logical_or"
#> [267,] "logistic_ccdf_log"           "logistic_cdf"
#> [268,] "logistic_cdf_log"            "logistic_lccdf"
#> [269,] "logistic_lcdf"               "logistic_log"
#> [270,] "logistic_lpdf"               "logistic_rng"
#> [271,] "logit"                       "loglogistic_cdf"
#> [272,] "loglogistic_log"             "loglogistic_lpdf"
#> [273,] "loglogistic_rng"             "lognormal_ccdf_log"
#> [274,] "lognormal_cdf"               "lognormal_cdf_log"
#> [275,] "lognormal_lccdf"             "lognormal_lcdf"
#> [276,] "lognormal_log"               "lognormal_lpdf"
#> [277,] "lognormal_rng"               "lub_constrain"
#> [278,] "lub_free"                    "make_iter_name"
#> [279,] "make_nu"                     "map_rect"
#> [280,] "map_rect_combine"            "map_rect_concurrent"
#> [281,] "map_rect_mpi"                "map_rect_reduce"
#> [282,] "matrix_exp"                  "matrix_exp_2x2"
#> [283,] "matrix_exp_action_handler"   "matrix_exp_multiply"
#> [284,] "matrix_exp_pade"             "matrix_normal_prec_log"
#> [285,] "matrix_normal_prec_lpdf"     "matrix_normal_prec_rng"
```

```
#> [286,] "matrix_power"                     "max"
#> [287,] "max_size"                         "max_size_mvt"
#> [288,] "mdivide_left"                     "mdivide_left_ldlt"
#> [289,] "mdivide_left_spd"                 "mdivide_left_tri"
#> [290,] "mdivide_left_tri_low"             "mdivide_right"
#> [291,] "mdivide_right_ldlt"               "mdivide_right_spd"
#> [292,] "mdivide_right_tri"                "mdivide_right_tri_low"
#> [293,] "mean"                             "min"
#> [294,] "minus"                            "modified_bessel_first_kind"
#> [295,] "modified_bessel_second_kind"      "modulus"
#> [296,] "mpi_cluster"                      "mpi_command"
#> [297,] "mpi_distributed_apply"            "mpi_parallel_call"
#> [298,] "multi_gp_cholesky_log"            "multi_gp_cholesky_lpdf"
#> [299,] "multi_gp_log"                     "multi_gp_lpdf"
#> [300,] "multi_normal_cholesky_log"        "multi_normal_cholesky_lpdf"
#> [301,] "multi_normal_cholesky_rng"        "multi_normal_log"
#> [302,] "multi_normal_lpdf"                "multi_normal_prec_log"
#> [303,] "multi_normal_prec_lpdf"           "multi_normal_prec_rng"
#> [304,] "multi_normal_rng"                 "multi_student_t_cholesky_lpdf"
#> [305,] "multi_student_t_cholesky_rng"     "multi_student_t_log"
#> [306,] "multi_student_t_lpdf"             "multi_student_t_rng"
#> [307,] "multinomial_log"                  "multinomial_logit_log"
#> [308,] "multinomial_logit_lpmf"           "multinomial_logit_rng"
#> [309,] "multinomial_lpmf"                 "multinomial_rng"
#> [310,] "multiply"                         "multiply_log"
#> [311,] "multiply_lower_tri_self_transpose" "neg_binomial_2_ccdf_log"
#> [312,] "neg_binomial_2_cdf"               "neg_binomial_2_cdf_log"
#> [313,] "neg_binomial_2_lccdf"             "neg_binomial_2_lcdf"
#> [314,] "neg_binomial_2_log"               "neg_binomial_2_log_glm_log"
#> [315,] "neg_binomial_2_log_glm_lpmf"      "neg_binomial_2_log_log"
#> [316,] "neg_binomial_2_log_lpmf"          "neg_binomial_2_log_rng"
#> [317,] "neg_binomial_2_lpmf"              "neg_binomial_2_rng"
#> [318,] "neg_binomial_ccdf_log"            "neg_binomial_cdf"
#> [319,] "neg_binomial_cdf_log"             "neg_binomial_lccdf"
#> [320,] "neg_binomial_lcdf"                "neg_binomial_log"
#> [321,] "neg_binomial_lpmf"                "neg_binomial_rng"
#> [322,] "norm"                             "norm1"
#> [323,] "norm2"                            "normal_ccdf_log"
```

```
#> [324,] "normal_cdf"                  "normal_cdf_log"
#> [325,] "normal_id_glm_log"           "normal_id_glm_lpdf"
#> [326,] "normal_lccdf"                "normal_lcdf"
#> [327,] "normal_log"                  "normal_lpdf"
#> [328,] "normal_rng"                  "normal_sufficient_log"
#> [329,] "normal_sufficient_lpdf"      "num_elements"
#> [330,] "ode_ckrk"                    "ode_rk45"
#> [331,] "ode_store_sensitivities"     "offset_multiplier_constrain"
#> [332,] "offset_multiplier_free"      "one_hot_array"
#> [333,] "one_hot_int_array"           "one_hot_row_vector"
#> [334,] "one_hot_vector"              "ones_array"
#> [335,] "ones_int_array"              "ones_row_vector"
#> [336,] "ones_vector"                 "operands_and_partials"
#> [337,] "operator_addition"           "operator_division"
#> [338,] "operator_equal_equal"        "operator_minus"
#> [339,] "operator_multiplication"     "operator_not_equal"
#> [340,] "operator_plus"               "operator_subtraction"
#> [341,] "ordered_constrain"           "ordered_free"
#> [342,] "ordered_logistic_glm_lpmf"   "ordered_logistic_log"
#> [343,] "ordered_logistic_lpmf"       "ordered_logistic_rng"
#> [344,] "ordered_probit_log"          "ordered_probit_lpmf"
#> [345,] "ordered_probit_rng"          "out_of_range"
#> [346,] "owens_t"                     "pareto_ccdf_log"
#> [347,] "pareto_cdf"                  "pareto_cdf_log"
#> [348,] "pareto_lccdf"                "pareto_lcdf"
#> [349,] "pareto_log"                  "pareto_lpdf"
#> [350,] "pareto_rng"                  "pareto_type_2_ccdf_log"
#> [351,] "pareto_type_2_cdf"           "pareto_type_2_cdf_log"
#> [352,] "pareto_type_2_lccdf"         "pareto_type_2_lcdf"
#> [353,] "pareto_type_2_log"           "pareto_type_2_lpdf"
#> [354,] "pareto_type_2_rng"           "partials_propagator"
#> [355,] "partials_return_type"        "partials_type"
#> [356,] "plain_type"                  "plus"
#> [357,] "poisson_binomial_ccdf_log"   "poisson_binomial_cdf"
#> [358,] "poisson_binomial_cdf_log"    "poisson_binomial_lccdf"
#> [359,] "poisson_binomial_lcdf"       "poisson_binomial_log"
#> [360,] "poisson_binomial_log_probs"  "poisson_binomial_lpmf"
#> [361,] "poisson_binomial_rng"        "poisson_ccdf_log"
```

```
#> [362,] "poisson_cdf"                "poisson_cdf_log"
#> [363,] "poisson_lccdf"              "poisson_lcdf"
#> [364,] "poisson_log"                "poisson_log_glm_log"
#> [365,] "poisson_log_glm_lpmf"       "poisson_log_log"
#> [366,] "poisson_log_lpmf"           "poisson_log_rng"
#> [367,] "poisson_lpmf"               "poisson_rng"
#> [368,] "polar"                      "positive_constrain"
#> [369,] "positive_free"              "positive_ordered_constrain"
#> [370,] "positive_ordered_free"      "possibly_sum"
#> [371,] "pow"                        "primitive_value"
#> [372,] "prob_constrain"             "prob_free"
#> [373,] "prod"                       "proj"
#> [374,] "promote_args"               "promote_elements"
#> [375,] "promote_scalar"             "promote_scalar_type"
#> [376,] "pseudo_eigenvalues"         "pseudo_eigenvectors"
#> [377,] "qr"                         "qr_Q"
#> [378,] "qr_R"                       "qr_thin"
#> [379,] "qr_thin_Q"                  "qr_thin_R"
#> [380,] "quad_form"                  "quad_form_diag"
#> [381,] "quad_form_sym"              "quantile"
#> [382,] "rank"                       "rayleigh_ccdf_log"
#> [383,] "rayleigh_cdf"               "rayleigh_cdf_log"
#> [384,] "rayleigh_lccdf"             "rayleigh_lcdf"
#> [385,] "rayleigh_log"               "rayleigh_lpdf"
#> [386,] "rayleigh_rng"               "read_corr_L"
#> [387,] "read_corr_matrix"           "read_cov_L"
#> [388,] "read_cov_matrix"            "real"
#> [389,] "reduce_sum"                 "reduce_sum_static"
#> [390,] "ref_type"                   "rep_array"
#> [391,] "rep_matrix"                 "rep_row_vector"
#> [392,] "rep_vector"                 "require_generics"
#> [393,] "require_helpers"            "resize"
#> [394,] "return_type"                "reverse"
#> [395,] "rising_factorial"           "round"
#> [396,] "row"                        "rows"
#> [397,] "rows_dot_product"           "rows_dot_self"
#> [398,] "scalar_seq_view"            "scalar_type"
#> [399,] "scalar_type_pre"            "scalbn"
```

```
#> [400,] "scale_matrix_exp_multiply"          "scaled_add"
#> [401,] "scaled_inv_chi_square_ccdf_log"     "scaled_inv_chi_square_cdf"
#> [402,] "scaled_inv_chi_square_cdf_log"      "scaled_inv_chi_square_lccdf"
#> [403,] "scaled_inv_chi_square_lcdf"         "scaled_inv_chi_square_log"
#> [404,] "scaled_inv_chi_square_lpdf"         "scaled_inv_chi_square_rng"
#> [405,] "sd"                                 "segment"
#> [406,] "select"                             "seq_view"
#> [407,] "sign"                               "signbit"
#> [408,] "simplex_constrain"                  "simplex_free"
#> [409,] "sin"                                "singular_values"
#> [410,] "sinh"                               "size"
#> [411,] "size_mvt"                           "size_zero"
#> [412,] "skew_double_exponential_ccdf_log"   "skew_double_exponential_cdf"
#> [413,] "skew_double_exponential_cdf_log"    "skew_double_exponential_lccdf"
#> [414,] "skew_double_exponential_lcdf"       "skew_double_exponential_log"
#> [415,] "skew_double_exponential_lpdf"       "skew_double_exponential_rng"
#> [416,] "skew_normal_ccdf_log"               "skew_normal_cdf"
#> [417,] "skew_normal_cdf_log"                "skew_normal_lccdf"
#> [418,] "skew_normal_lcdf"                   "skew_normal_log"
#> [419,] "skew_normal_lpdf"                   "skew_normal_rng"
#> [420,] "softmax"                            "sort_asc"
#> [421,] "sort_desc"                          "sort_indices"
#> [422,] "sort_indices_asc"                   "sort_indices_desc"
#> [423,] "sqrt"                               "square"
#> [424,] "squared_distance"                   "stan_print"
#> [425,] "static_select"                      "std_normal_ccdf_log"
#> [426,] "std_normal_cdf"                     "std_normal_cdf_log"
#> [427,] "std_normal_lccdf"                   "std_normal_lcdf"
#> [428,] "std_normal_log"                     "std_normal_log_qf"
#> [429,] "std_normal_lpdf"                    "std_normal_rng"
#> [430,] "step"                               "student_t_ccdf_log"
#> [431,] "student_t_cdf"                      "student_t_cdf_log"
#> [432,] "student_t_lccdf"                    "student_t_lcdf"
#> [433,] "student_t_log"                      "student_t_lpdf"
#> [434,] "student_t_rng"                      "sub_col"
#> [435,] "sub_row"                            "subtract"
#> [436,] "sum"                                "svd"
#> [437,] "svd_U"                              "svd_V"
```

```
#> [438,] "symmetrize_from_lower_tri"        "symmetrize_from_upper_tri"
#> [439,] "system_error"                     "tail"
#> [440,] "tan"                              "tanh"
#> [441,] "tcrossprod"                       "tgamma"
#> [442,] "throw_domain_error"               "throw_domain_error_mat"
#> [443,] "throw_domain_error_vec"           "to_array_1d"
#> [444,] "to_array_2d"                      "to_complex"
#> [445,] "to_int"                           "to_matrix"
#> [446,] "to_ref"                           "to_row_vector"
#> [447,] "to_vector"                        "trace"
#> [448,] "trace_gen_inv_quad_form_ldlt"     "trace_gen_quad_form"
#> [449,] "trace_inv_quad_form_ldlt"         "trace_quad_form"
#> [450,] "transpose"                        "trigamma"
#> [451,] "trunc"                            "typedefs"
#> [452,] "ub_constrain"                     "ub_free"
#> [453,] "uniform_ccdf_log"                 "uniform_cdf"
#> [454,] "uniform_cdf_log"                  "uniform_lccdf"
#> [455,] "uniform_lcdf"                     "uniform_log"
#> [456,] "uniform_lpdf"                     "uniform_rng"
#> [457,] "uniform_simplex"                  "unit_vector_constrain"
#> [458,] "unit_vector_free"                 "unitspaced_array"
#> [459,] "validate_non_negative_index"      "validate_positive_index"
#> [460,] "validate_unit_vector_index"       "value_of"
#> [461,] "value_of_rec"                     "value_type"
#> [462,] "variance"                         "vec_concat"
#> [463,] "vector_seq_view"                  "void_t"
#> [464,] "von_mises_ccdf_log"               "von_mises_cdf"
#> [465,] "von_mises_cdf_log"                "von_mises_lccdf"
#> [466,] "von_mises_lcdf"                   "von_mises_log"
#> [467,] "von_mises_lpdf"                   "von_mises_rng"
#> [468,] "weibull_ccdf_log"                 "weibull_cdf"
#> [469,] "weibull_cdf_log"                  "weibull_lccdf"
#> [470,] "weibull_lcdf"                     "weibull_log"
#> [471,] "weibull_lpdf"                     "weibull_rng"
#> [472,] "welford_covar_estimator"          "welford_var_estimator"
#> [473,] "wiener_log"                       "wiener_lpdf"
#> [474,] "wishart_cholesky_lpdf"            "wishart_cholesky_rng"
#> [475,] "wishart_log"                      "wishart_lpdf"
```

```
#> [476,] "wishart_rng"          "zeros_array"
#> [477,] "zeros_int_array"      "zeros_row_vector"
#> [478,] "zeros_vector"         ""
```

Using Higher-Order Functions in the StanHeaders Package

This section will demonstrate how to use some of the C++ functions in the **StanHeaders** package whose first argument is another C++ function, in which case the `stanFunction` in the previous section will not work and you have to write your own C++.

Derivatives and Minimization

The following is a toy example of using the Stan Math library via `Rcpp::sourceCpp`: to minimize the function

$$(\mathbf{x} - \mathbf{a})^\top (\mathbf{x} - \mathbf{a})$$

which has a global minimum when $\mathbf{x} = \mathbf{a}$. To find this minimum with autodifferentiation, we need to define the objective function. Then, its gradient with respect to $\mathbf{x}$, which we know is $2(\mathbf{x} - \mathbf{a})$ in this case, can be calculated by autodifferentiation. At the optimum (or on the way to the optimum), we might want to evaluate the Hessian matrix, which we know is $2\mathbf{I}$, but would need an additional function to evaluate it via autodifferentiation. Finally, one could reconceptualize the problem as solving a homogeneous system of equations where the gradient is set equal to a vector of zeros. The `stan::math::algebra_solver` function can solve such a system using autodifferentiation to obtain the Jacobian, which we know to be the identity matrix in this case.

```
Sys.setenv(PKG_CXXFLAGS = StanHeaders:::CxxFlags(as_character = TRUE))
SH <- system.file(ifelse(.Platform$OS.type == "windows", "libs", "lib"), .Platform$r_arch,
                  package = "StanHeaders", mustWork = TRUE)
Sys.setenv(PKG_LIBS = paste0(StanHeaders:::LdFlags(as_character = TRUE),
                             " -L", shQuote(SH), " -lStanHeaders"))
```

Here is C++ code that does all of the above, except for the part of finding the optimum, which is done using the R function `optim` below.

```
// [[Rcpp::depends(BH)]]
// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::depends(RcppParallel)]]
// [[Rcpp::depends(StanHeaders)]]
#include <stan/math/mix.hpp> // stuff from mix/ must come first
#include <stan/math.hpp>     // finally pull in everything from rev/ and prim/
#include <Rcpp.h>
#include <RcppEigen.h>       // do this AFTER including stuff from stan/math

// [[Rcpp::plugins(cpp17)]]

/* Objective function */

// [[Rcpp::export]]
auto f(Eigen::VectorXd x, Eigen::VectorXd a) { // objective function in doubles
  using stan::math::dot_self;                  // dot_self() is a dot product with self
  return dot_self( (x - a) );
}

/* Gradient */

// [[Rcpp::export]]
auto g(Eigen::VectorXd x, Eigen::VectorXd a) {  // gradient by AD using Stan
  double fx;
  Eigen::VectorXd grad_fx;
  using stan::math::dot_self;
  stan::math::gradient([&a](auto x) { return dot_self( (x - a) ); },
                       x, fx, grad_fx);
  return grad_fx;
}

/* Hessian */

// [[Rcpp::export]]
```

```
auto H(Eigen::VectorXd x, Eigen::VectorXd a) { // Hessian by AD using Stan
  double fx;
  Eigen::VectorXd grad_fx;
  Eigen::MatrixXd H;
  using stan::math::dot_self;
  stan::math::hessian([&a](auto x) { return dot_self(x - a); },
                      x, fx, grad_fx, H);
  return H;
}

/* Jacobian */

// [[Rcpp::export]]
auto J(Eigen::VectorXd x, Eigen::VectorXd a) { // not actually used
  Eigen::VectorXd fx;
  Eigen::MatrixXd J;
  using stan::math::dot_self;
  stan::math::jacobian([&a](auto x) {
    return (2 * (x - a));
  }, x, fx, J);
  return J;
}

struct equations_functor {
  template <typename T0, typename T1>
  inline Eigen::Matrix<T0, Eigen::Dynamic, 1>
  operator()(const Eigen::Matrix<T0, Eigen::Dynamic, 1>& x,
             const Eigen::Matrix<T1, Eigen::Dynamic, 1>& theta,
             const std::vector<double>& x_r, const std::vector<int>& x_i,
             std::ostream* pstream__) const {
    return 2 * (x - stan::math::to_vector(x_r));
  }
};

// [[Rcpp::export]]
auto solution(Eigen::VectorXd a, Eigen::VectorXd guess) {
  Eigen::VectorXd theta;
  auto x_r = stan::math::to_array_1d(a);
```

```
  equations_functor f;
  auto x = stan::math::algebra_solver(f, guess, theta, x_r, {});
  return x;
}
```

In this compiled RMarkdown document, the **knitr** package has exported functions `f`, `g`, `H`, `J` and `solution` (but not `equations_functor`) to R's global environment using the `sourceCpp` function in the **Rcpp** package, so that they can now be called from R. Here we find the optimum starting from a random point in three dimensions:

```
x <- optim(rnorm(3), fn = f, gr = g, a = 1:3, method = "BFGS", hessian = TRUE)
x$par
#> [1] 1 2 3
x$hessian
#>      [,1] [,2] [,3]
#> [1,]    2    0    0
#> [2,]    0    2    0
#> [3,]    0    0    2
H(x$par, a = 1:3)
#>      [,1] [,2] [,3]
#> [1,]    2    0    0
#> [2,]    0    2    0
#> [3,]    0    0    2
J(x$par, a = 1:3)
#>      [,1] [,2] [,3]
#> [1,]    2    0    0
#> [2,]    0    2    0
#> [3,]    0    0    2
solution(a = 1:3, guess = rnorm(3))
#> [1] 1 2 3
```

Integrals and Ordinary Differential Equations

The Stan Math library can do one-dimensional numerical integration and can solve stiff and non-stiff systems of differential equations, such as the harmonic oscillator example below. Solving stiff systems utilizes the CVODES library, which is included in **StanHeaders**.

```
// [[Rcpp::depends(BH)]]
// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::depends(RcppParallel)]]
// [[Rcpp::depends(StanHeaders)]]
#include <stan/math.hpp>                        // pulls in everything from rev/ and prim/
#include <Rcpp.h>
#include <RcppEigen.h>                          // do this AFTER including stan/math

// [[Rcpp::plugins(cpp17)]]

/* Definite integrals */

// [[Rcpp::export]]
double Cauchy(double scale) {
  std::vector<double> theta;
  auto half = stan::math::integrate_1d([](auto x, auto xc, auto theta,
                                          auto x_r, auto x_i, auto msgs) {
    return exp(stan::math::cauchy_lpdf(x, 0, x_r[0]));
  }, -scale, scale, theta, {scale}, {}, nullptr, 1e-7);
  return half * 2; // should equal 1 for any positive scale
}

/* Ordinary Differential Equations */

// [[Rcpp::export]]
auto nonstiff(Eigen::MatrixXd A, Eigen::VectorXd y0) {
  using stan::math::integrate_ode_rk45;
  using stan::math::to_vector;
  using stan::math::to_array_1d;
  std::vector<double> theta;
  std::vector<double> times = {1, 2};
  auto y = integrate_ode_rk45([&A](auto t, auto y,
                                   auto theta, auto x_r, auto x_i, std::ostream *msgs) {
```

```
    return to_array_1d( (A * to_vector(y)).eval() );
  }, to_array_1d(y0), 0, times, theta, {}, {});
  Eigen::VectorXd truth = stan::math::matrix_exp(A) * y0;
  return (to_vector(y[0]) - truth).eval(); // should be "zero"
}

// [[Rcpp::export]]
auto stiff(Eigen::MatrixXd A, Eigen::VectorXd y0) { // not actually stiff
  using stan::math::integrate_ode_bdf;              // but use the stiff solver anyways
  using stan::math::to_vector;
  using stan::math::to_array_1d;
  std::vector<double> theta;
  std::vector<double> times = {1, 2};
  auto y = integrate_ode_bdf([&A](auto t, auto y,
                                  auto theta, auto x_r, auto x_i, std::ostream *msgs) {
    return to_array_1d( (A * to_vector(y)).eval() );
  }, to_array_1d(y0), 0, times, theta, {}, {});
  Eigen::VectorXd truth = stan::math::matrix_exp(A) * y0;
  return (to_vector(y[0]) - truth).eval(); // should be "zero"
}
```

Again, in this compiled RMarkdown document, the **knitr** package has exported the `Cauchy`, `nonstiff` and `stiff` functions to R's global environment using the `sourceCpp` function in the **Rcpp** package so that they can be called from R.

First, we numerically integrate the Cauchy PDF over its interquartile range — which has an area of $\frac{1}{2}$ — that we then double to verify that it is almost within machine precision of 1.

```
all.equal(1, Cauchy(rexp(1)), tol = 1e-15)
#> [1] TRUE
```

Next, we consider the system of differential equations

$$\frac{d}{dt}\mathbf{y} = \mathbf{A}\mathbf{y}$$

where $\mathbf{A}$ is a square matrix such as that for a simple harmonic oscillator

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -1 & -\theta \end{bmatrix}$$

for $\theta \in (0, 1)$. The solution for $\mathbf{y}_t = e^{t\mathbf{A}}\mathbf{y}_0$ can be obtained via the matrix exponential function, which is available in the Stan Math Library, but it can also be obtained numerically using a fourth-order Runge-Kutta solver, which is appropriate for non-stiff systems of ODEs, such as this one. However, it is possible, albeit less efficient in this case, to use the backward-differentiation formula solver for stiff systems of ODEs. In both cases, we calculate the difference between the analytical solution and the numerical one, and the stiff version does produce somewhat better accuracy in this case.

```
A <- matrix(c(0, -1, 1, -runif(1)), nrow = 2, ncol = 2)
y0 <- rexp(2)
all.equal(nonstiff(A, y0), c(0, 0), tol = 1e-5)
#> [1] TRUE
all.equal(   stiff(A, y0), c(0, 0), tol = 1e-8)
#> [1] TRUE
```

Parellelization

`map_rect` Function

The Stan Math Library includes the `map_rect` function, which applies a function to each element of rectangular arrays and returns a vector, making it a bit like a restricted version of R's `sapply` function. However, `map_rect` can also be executed in parallel by defining the pre-processor directive `STAN_THREADS` and then setting the `STAN_NUM_THREADS` environmental variable to be the number of threads to use, as in

```
Sys.setenv(STAN_NUM_THREADS = 2) # specify -1 to use all available cores
```

Below is C++ code to test whether an integer is prime, using a rather brute-force algorithm and running it in parallel via `map_rect`.

```
// [[Rcpp::depends(BH)]]
// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::depends(RcppParallel)]]
```

```
// [[Rcpp::depends(StanHeaders)]]
#include <stan/math.hpp>                    // pulls in everything from rev/ and prim/
#include <Rcpp.h>
#include <RcppEigen.h>                      // do this AFTER including stan/math

// [[Rcpp::plugins(cpp17)]]

// see https://en.wikipedia.org/wiki/Primality_test#Pseudocode
struct is_prime {
  is_prime() {}
  template <typename T1, typename T2>
  auto
  operator()(const Eigen::Matrix<T1, Eigen::Dynamic, 1>& eta,
             const Eigen::Matrix<T2, Eigen::Dynamic, 1>& theta,
             const std::vector<double>& x_r, const std::vector<int>& x_i,
             std::ostream* msgs = nullptr) const {
    Eigen::VectorXd res(1); // can only return double or var vectors
    int n = x_i[0];
    if (n <= 3) {
      res.coeffRef(0) = n > 1;
      return res;
    } else if ( (n % 2 == 0) || (n % 3 == 0) ) {
      res.coeffRef(0) = false;
      return res;
    }
    int i = 5;
    while (i * i <= n) {
      if ( (n % i == 0) || (n % (i + 2) == 0) ) {
        res.coeffRef(0) = false;
        return res;
      }
      i += 6;
    }
    res.coeffRef(0) = true;
    return res;
  }
};
```

```
/* parallelization */
// [[Rcpp::export]]
auto psapply(std::vector<std::vector<int> > n) {
  std::vector<Eigen::VectorXd> eta(n.size()); // these all have to be the same size
  Eigen::VectorXd theta;
  std::vector<std::vector<double> > x_d(n.size());
  return stan::math::map_rect<0, is_prime>(theta, eta, x_d, n, &Rcpp::Rcout);
}
```

Since the signature for `n` is a `std::vector<std::vector<int> >`, we have to pass it from R as a list (which is converted to the outer `std::vector<>`) of integer vectors (which is converted to the inner `std::vector<int>`) that happen to be of size one in this case.

```
odd <- seq.int(from = 2^25 - 1, to = 2^26 - 1, by = 2)
tail(psapply(n = as.list(odd))) == 1 # check your process manager while this is running
#> [1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

Thus, $2^{26} - 5 = 67,108,859$ is a prime number.

`reduce_sum` Function

The `reduce_sum` function can be used to sum a function of recursively-partitioned data in parallel. The Probability Mass Function (PMF) of the logarithmic distribution is

$$\Pr(x = k \mid p) = \frac{p^k}{-k \ln(1 - p)}$$

for a positive integer k and $0 < p < 1$. To verify that this PMF sums to 1 over the natural numbers, we could accumulate it for a large finite number of terms, but would like to do so in parallel. To do so, we need to define a `partial_sum` function that adds the PMF for some subset of natural numbers and pass it to the `reduce_sum` function like

```
// [[Rcpp::depends(BH)]]
// [[Rcpp::depends(RcppEigen)]]
// [[Rcpp::depends(RcppParallel)]]
```

```
// [[Rcpp::depends(StanHeaders)]]
#include <stan/math.hpp>                    // pulls in everything from rev/ and prim/
#include <Rcpp.h>
#include <RcppEigen.h>                      // do this AFTER including stan/math

// [[Rcpp::plugins(cpp17)]]

template <typename T>
struct partial_sum {
  partial_sum() {}
  T operator()(const std::vector<int>& k_slice,
               int start, int end, // these are ignored in this example
               std::ostream* msgs, double p) {
    double S = 0;
    for (int n = 0; n < k_slice.size(); n++) {
      int k = k_slice[n];
      S += stan::math::pow(p, k) / k;
    }
    return S;
  }
};

// [[Rcpp::export]]
double check_logarithmic_PMF(const double p, const int N = 1000) {
  using stan::math::reduce_sum;
  std::vector<int> k(N);
  std::iota(k.begin(), k.end(), 1);
  return reduce_sum<partial_sum<double> >(k, 1, nullptr, p) / -std::log1p(-p);
}
```

The second argument passed to `reduce_sum` is the `grainsize`, which governs the size (and number) of the recursive subsets of `k` that are passed to the `partial_sum` function. A `grainsize` of 1 indicates that the software will try to automatically tune the subset size for good performance, but that may be worse than choosing a `grainsize` by hand in a problem-specific fashion.

In any event, we can call the wrapper function `check_logarithmic_PMF` in R to verify that it returns 1 to numerical tolerance:

```
stopifnot(all.equal(1, check_logarithmic_PMF(p = 1 / sqrt(2))))
```

Defining a Stan Model in C++

The Stan *language* does not have much support for sparse matrices for a variety of reasons. Essentially the only applicable function is `csr_matrix_times_vector`, which pre-multiplies a vector by a sparse matrix in compressed row storage by taking as arguments its number of rows, columns, non-zero values, column indices of non-zero values, and locations where the non-zero values start in each row. While the `csr_matrix_times_vector` function could be used to implement the example below, we illustrate how to use the sparse data structures in the **Matrix** and **RcppEigen** packages in a Stan model written in C++, which could easily be extended to more complicated models with sparse data structures.

Our C++ file for the log-likelihood of a linear model with a sparse design matrix reads as

```
#include <stan/model/model_header.hpp>
#include <Rcpp.h>
#include <RcppEigen.h>

class sparselm_stan {

public: // these would ordinarily be private in the C++ code generated by Stan
  Eigen::Map<Eigen::SparseMatrix<double> > X;
  Eigen::VectorXd y;

  sparselm_stan(Eigen::Map<Eigen::SparseMatrix<double> > X, Eigen::VectorXd y) :
    X(X), y(y) {}

  template <bool propto__ = false, bool jacobian__ = false, typename T__ = double>
  // propto__ is usually true but causes log_prob() to return 0 when called from R
  // jacobian__ is usually true for MCMC but typically is false for optimization
  T__ log_prob(std::vector<T__>& params_r__) const {
    using namespace stan::math;
    T__ lp__(0.0);
    accumulator<T__> lp_accum__;

    // set up model parameters
    std::vector<int> params_i__;
```

```
    stan::io::deserializer<T__> in__(params_r__, params_i__);
    auto beta = in__.template read<Eigen::Matrix<T__,-1,1> >(X.cols());
    auto sigma = in__.template read_constrain_lb<T__, jacobian__>(0, lp__);

    // log-likelihood (should add priors)
    lp_accum__.add(lp__);
    lp_accum__.add(normal_lpdf<propto__>(y, (X * beta).eval(), sigma));
    return lp_accum__.sum();
  }

  template <bool propto__ = false, bool jacobian__ = false>
  std::vector<double> gradient(std::vector<double>& params_r__) const {
    // Calculate gradients using reverse-mode autodiff
    // although you could do them analytically in this case

    using std::vector;
    using stan::math::var;
    double lp;
    std::vector<double> gradient;
    try {
      vector<var> ad_params_r(params_r__.size());
      for (size_t i = 0; i < params_r__.size(); ++i) {
        var var_i(params_r__[i]);
        ad_params_r[i] = var_i;
      }
      var adLogProb
        = this->log_prob<propto__, jacobian__>(ad_params_r);
      lp = adLogProb.val();
      adLogProb.grad(ad_params_r, gradient);
    } catch (const std::exception &ex) {
      stan::math::recover_memory();
      throw;
    }
    stan::math::recover_memory();
    return gradient;
  }
};
```

To use it from R, we call the `exposeClass` function in the **Rcpp** package with the necessary arguments and then call `sourceCpp` on the file it wrote in the temporary directory:

```
library(Rcpp)
tf <- tempfile(fileext = "Module.cpp")
exposeClass("sparselm_stan",
      constructors = list(c("Eigen::Map<Eigen::SparseMatrix<double> >",
                            "Eigen::VectorXd")),
      fields = c("X", "y"),
      methods = c("log_prob<>", "gradient<>"),
      rename = c(log_prob = "log_prob<>", gradient = "gradient<>"),
      header = c("// [[Rcpp::depends(BH)]]",
                 "// [[Rcpp::depends(RcppEigen)]]",
                 "// [[Rcpp::depends(RcppParallel)]",
                 "// [[Rcpp::depends(StanHeaders)]]",
                 "// [[Rcpp::plugins(cpp17)]]",
                 paste0("#include <", file.path(getwd(), "sparselm_stan.hpp"), ">")),
      file = tf,
      Rfile = FALSE)
Sys.setenv(PKG_CXXFLAGS = paste0(Sys.getenv("PKG_CXXFLAGS"), " -I",
                                 system.file("include", "src",
                                             package = "StanHeaders", mustWork = TRUE)))
sourceCpp(tf)
sparselm_stan
#> C++ class 'sparselm_stan' <0x557ae52e0380>
#> Constructors:
#>     sparselm_stan(Eigen::Map<Eigen::SparseMatrix<double, 0, int>, 0, Eigen::Stride<0, 0> >,
  Eigen::Matrix<double, -1, 1, 0, -1, 1>)
#>
#> Fields:
#>     Eigen::Map<Eigen::SparseMatrix<double, 0, int>, 0, Eigen::Stride<0, 0> > X
#>     Eigen::Matrix<double, -1, 1, 0, -1, 1> y
#>
#> Methods:
#>      std::vector<double, std::allocator<double> > gradient(std::vector<double,
  std::allocator<double> >)  const
#>
```

```
#>      double log_prob(std::vector<double, std::allocator<double> >)  const
#>
```

At this point, we need a sparse design matrix and (dense) outcome vector to pass to the constructor. The former can be created with a variety of functions in the **Matrix** package, such as

```
dd <- data.frame(a = gl(3, 4), b = gl(4, 1, 12))
X <- Matrix::sparse.model.matrix(~ a + b, data = dd)
X
#> 12 x 6 sparse Matrix of class "dgCMatrix"
#>    (Intercept) a2 a3 b2 b3 b4
#> 1            1  .  .  .  .  .
#> 2            1  .  .  1  .  .
#> 3            1  .  .  .  1  .
#> 4            1  .  .  .  .  1
#> 5            1  1  .  .  .  .
#> 6            1  1  .  1  .  .
#> 7            1  1  .  .  1  .
#> 8            1  1  .  .  .  1
#> 9            1  .  1  .  .  .
#> 10           1  .  1  1  .  .
#> 11           1  .  1  .  1  .
#> 12           1  .  1  .  .  1
```

Finally, we call the `new` function in the **methods** package, which essentially calls our C++ constructor and provides an R interface to the instantiated object, which contains the `log_prob` and `gradient` methods we defined and can be called with arbitrary inputs.

```
sm <- new(sparselm_stan, X = X, y = rnorm(nrow(X)))
sm$log_prob(c(beta = rnorm(ncol(X)), log_sigma = log(pi)))
#> [1] -26.01577
round(sm$gradient(c(beta = rnorm(ncol(X)), log_sigma = log(pi))), digits = 4)
#> [1] -2.1984 -1.0424 -0.5089 -0.4552 -0.6047 -0.8303 -6.6551
```